COMPARISON OF PERFORMANCE OF ARTIFICIAL NEURAL NETWORK ON TYPICAL BENCHMARK PROBLEMS USING MATLAB TOOLS VIS-A-VIS CODES WRITTEN IN JAVA

By

LT CDR SUSHIL KUMAR
Roll No 9910481

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
JANUARY, 2001

COMPARISON OF PERFORMANCE OF ARTIFICIAL NEURAL NETWORK ON TYPICAL BENCHMARK PROBLEMS USING MATLAB TOOLS VIS-À-VIS CODES WRITTEN IN JAVA

Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY
IN COMMUNICATION

By

LT CDR SUSHIL KUMAR
Roll No 9910481

To the
ELECTRICAL DEPARTMENT/ACES
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JANUARY 2001

CERTIFICATE

This is to certify that the work contained in this thesis titled "Comparison of performance of Artificial Neural Network on Typical Benchmark Problems using MATLAB Tools vis-à-vis Codes written in Java" by Lt Cdr Sushil Kumar, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr P.K.Kalra)
Professor
Dept. of Elec. Engg.
IIT Kanpur

ABSTRACT

An attempt has been made in this project to find optimum neural network configuration, using MATLAB Toolbox, for some of the benchmark problems. These problems are considered difficult to solve, using standard ANN techniques. Beside these problems some complicated functions have also been considered and an attempt has been made to solve them. All the problems considered in this work are typical in the sense that they capture the extremities of most of the parameters. Problems considered fall in the category of Classification and Function Approximation.

Codes developed in JAVA were used to solve the same problems. The results thus obtained were used to compare to that obtained by using MATLAB Toolbox. The codes developed using JAVA have undergone refinement may be used for modeling of Neural Networks for real life problems.

ACKNOWLEDGEMENT

I wish to express my deepest sense of gratitude and sincere thanks to Dr P K Kalra, my thesis supervisor, for introducing me to the interesting world of Artificial Neural Networks and Artificial Intelligence. The transparent working atmosphere, never ending exchange of ideas and his constant encouragement was the main driving force throughout the course of this study.

Steel Authority of India and BARC, Mumbai are hereby acknowledged for their financial support to this project.

I would also like to express my thanks to Mr A Prashant, who was always there to help, discuss and share his experience in this field. I am grateful to Miss Manu Aggarwal and Miss Amboo Srivastava for their help. Last but not the least, I would like to thank my wife, Sonia, for her constant encouragement during the tough times.

Lt Cdr Sushil Kumar

Jan 2001

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Several attempts have been reported to understand and model the capabilities of human brain. Some of these are *Expert System, Genetic Algorithm, Artificial Neural Networks, Fuzzy Logic etc.* These algorithms represent different level of human information processing. It has been believed that human brain has exceptional capability to *recall, optimize, memorize, sort and search.* However, each model may not perform all functions performed by brain independently. In the present work the computational power of Artificial Neural Networks have been explored.

1.1 Artificial Neural Networks

ANN consists of many simple elements called *neurons.* The neurons interact with each other using weighted connection similar to **biological neurons**. Inputs to artificial neural net are multiplied by corresponding weights. All the weighted inputs are then segregated and then subjected to non-linear filtering to determine the state or active level of the neurons.

Neurons are generally configured in regular and highly interconnected topology in ANN. For example, in the Hopfield model, neurons form a fully connected topology, and output from each neuron feeds as an input to the neighboring neurons. In the Backpropagation model and Boltzman machine, the networks consist of one or more layers between input and output layers. In the self-organizing feature map, the networks connect a vector of input neurons to a two dimensional grid of output neurons. There is no clear cut methodology to decide parameters, topologies and method of training of ANN. Hence, to build the ANN is time consuming and computer intensive. However these can be used in real time because of *inherent parallelisms and noise immunity* characteristics.

1.2 Models of Artificial Neural Network

Neural networks can be defined as an interconnection of neurons such that neuron outputs are connected, through weights, to all other neurons including themselves; both lag-free and delay connections are permitted. The networks can be broadly classified as:

(i) Feed forward Network

(ii) Feed back Network

1.2.1 **Feed forward Network:** Considering an elementary feed-forward architecture of m neurons receiving n inputs as shown in the next figure.

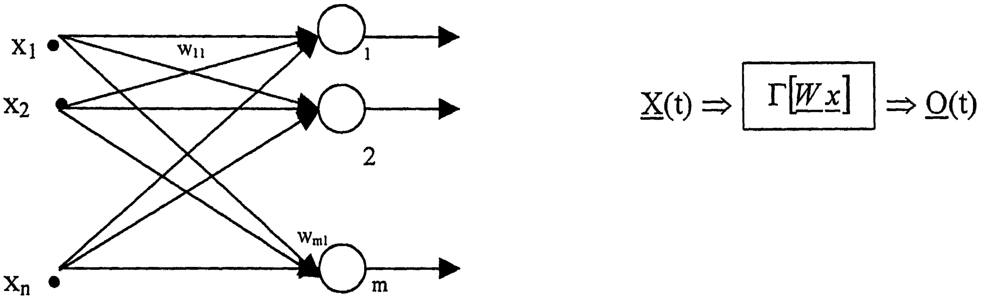


Figure 1.1: A feed-forward network

m = number of neurons,

n = number of inputs

Its output and input vectors are given by:

Input Vector $\underline{X}=[x_1,x_2 \dots\dots\dots\dots x_n]^t$

Output Vector $\underline{O}=[0_1,0_2 \dots\dots\dots\dots 0_m]^t$ (1.1)

w_{ij} = weight between i^{th} neuron and j^{th} input

Activation value for i^{th} neuron can be written as

$$net_i = \sum_{j=1}^{n} w_{ij} x_j \text{, for i = 1, 2.................m} \qquad (1.2)$$

The non-linear transformation after this is

$$O_i = f(\underline{w}_i^t \underline{x}) \text{; i = 1, 2................m}$$

The above-mentioned non-linear transformation is performed by each of the m neurons

$$O_i = f(\underline{w}_i^t \underline{x}), \quad \mathrm{i} = 1, 2.....................\mathrm{m} \tag{1.3}$$

$$\underline{W}_i \underline{\underline{\Delta}} \left[w_{i1}\ w_{i2} \ldots\ldots\ldots\ldots w_{in}\right]^t \tag{1.4}$$

Γ = Non-linear matrix operator, the mapping of input space x to output space 0 implemented by the network can be expressed as follows:

$$O = \Gamma\left[\underline{W}\,\underline{x}\right] \tag{1.5}$$

$\underline{W}$ = Weight matrix or connection matrix

$$\underline{W}\,\underline{\underline{\Delta}} \begin{bmatrix} w_{11} & w_{12} & \cdot & \cdot & w_{1n} \\ w_{21} & \cdot & \cdot & \cdot & w_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ w_{m1} & \cdot & \cdot & \cdot & w_{mn} \end{bmatrix} \tag{1.6a}$$

and

$$\underline{\Gamma\left[.\right]}\,\underline{\underline{\Delta}} \begin{bmatrix} f(.) & 0 & \cdot & \cdot & 0 \\ 0 & f(.) & \cdot & \cdot & 0 \\ 0 & \cdot & \cdot & \cdot & f(.) \end{bmatrix} \tag{1.6b}$$

f(.) = Non linear activation function lying on the diagonal of Γ operates component-wise on activation values (net) of each neuron.

$\underline{X}$ and $\underline{O}$ are often called input and output patterns. The mapping of an input pattern into an output pattern is of the feed forward and instantaneous type, since it involves no time delays between the input $\underline{X}$ and the output $\underline{O}$.

$$\therefore \qquad O(t) = \Gamma\left[\underline{W}\ \underline{x}\ (t)\right] \tag{1.7}$$

Example 1.1: Two-layer feed-forward network using neurons having bipolar binary activation function

$$f(net) \underset{=}{\Delta} \operatorname{sgn}(net) = \begin{cases} +1 & for\ net > 0 \\ -1 & for\ net < 0 \end{cases}$$

The network to be analyzed is shown below:

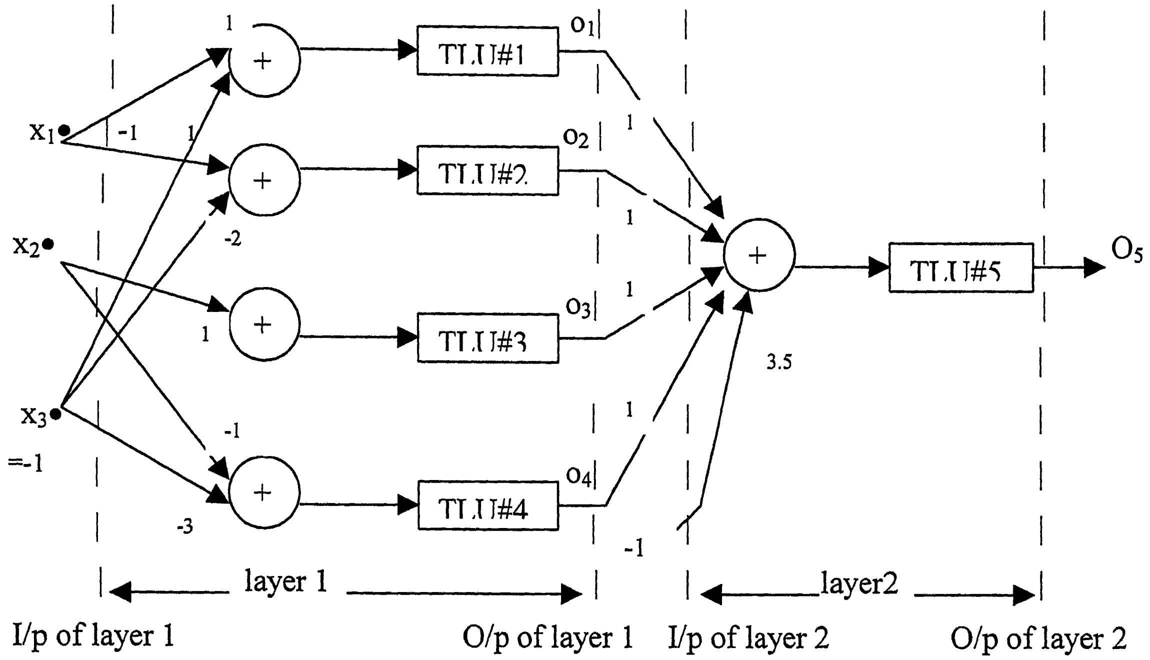


Fig1.2:A Two Layer Feed-forward Network

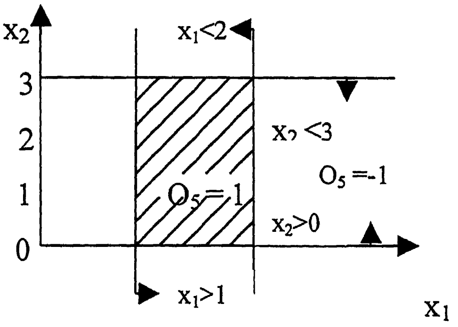


Fig 1.3: Two- Dimensional Space Mapping

Of the Network

Our purpose is to find output O$_5$ for a given network and input patterns. Each network is described by the formula:

$$\underline{O} = \Gamma\,[\underline{W}\ \underline{x}]$$

Consider the first layer:

$$\underline{O} = [O_1\,O_2\,O_3\,O_4]^t\ ; \quad \underline{x} = [x_1\,x_2\,x_3]^t \qquad \text{Or} \qquad [x_1\,x_2\,-1]^t$$

$$\underline{W}_1 = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & -1 & -3 \end{bmatrix}$$

Similarly for the second layer

$$\underline{O} = [O_5]\ ; \qquad \underline{x} = [O_1\ O_2\ O_3\ O_4\ -1]^t ; \qquad W_2 = [1\ \ 1\ \ 1\ \ 1\ \ 3.5]$$

The response of the first layer (for bipolar binary activation function) can be given as:

$$\underline{O} = [\operatorname{sgn}(x_1-1)\quad \operatorname{sgn}(2-x_1)\quad \operatorname{sgn}(x_2)\quad \operatorname{sgn}(-x_2+3)]^t$$

The mapping performed by the first layer is described next.

In this, each of the neurons 1 through 4 divides the plane x$_1$, x$_2$ into two half-planes. The half-planes where the neurons responses are positive (+1) have been marked on figure 1.3 with arrows pointing towards positive response half-plane. The response of the second layer can be easily obtained as:

$$O_5 = \operatorname{sgn}\ (O_1 + O_2 + O_3 + O_4 - 3.5)$$

$$O_5 = +1 \qquad \text{iff} \qquad O_1 = O_2 = O_3 = O_4 = 1$$

It therefore selects the intersection of four half-planes produced by the first layer and designated by the arrows.

Mapping Provided by the same Architecture but the Neurons having Sigmoidal Characteristic

For continuous bipolar activation function

$$f(net) = \frac{2}{1+\exp(-\lambda net)} - 1 \tag{1.8}$$

The output for first layer will be given by

$$\underline{O} = \begin{bmatrix} \frac{2}{1+\exp(1-x_1)\lambda} - 1 \\ \frac{2}{1+\exp(x_1-2)\lambda} - 1 \\ \frac{2}{1+\exp(-x_2)\lambda} - 1 \\ \frac{2}{1+\exp(x_2-3)\lambda} - 1 \end{bmatrix} \tag{1.9}$$

For the second layer, it would be

$$O_5 = \frac{2}{1+\exp(3.5-O_1-O_2-O_3-O_4)\lambda} - 1$$

The network with neurons having sigmoidal activation function can perform two-dimensional space mapping. The network considered with bipolar continuous neurons provides mapping of the entire x_1, x_2 plane into the interval (-1, 1) on the real number axis. In fact NN with as few as two layers are capable of universal approximation from one finite dimensional space to another

1.2.2 **Feed back Network**: By connecting the neurons' outputs to their inputs we can convert the feed-forward to feedback network as shown in figure 1.4. Feedback loop control the output O_i through outputs O_j , j =1, 2............. m. Here the present output $\underline{O}(t)$ controls the output at the following instant, $\underline{O}(t+\Delta)$. The time Δ' between t and t+Δ is introduced by the delay elements in the feedback loop

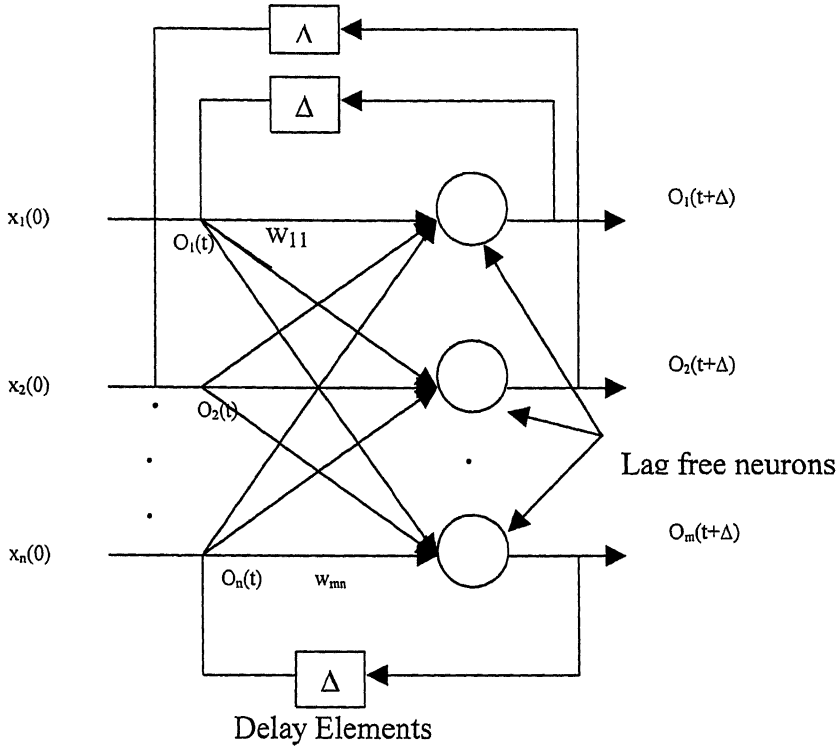


Fig 1.4: A feedback Network

$$\underline{O}\,(t+\Delta) \quad = \quad \Gamma[\underline{w}\;\underline{O}(t)] \qquad (1.10)$$

The input $\underline{x}(t)$ is only needed to initialize this network so that $\underline{O}(O)=\underline{x}(O)$. The input is then removed and the system remains autonomous for t > 0

There are two main categories of single layer feedback networks. If we consider time as a discrete variable and decide to observe the network performance at discrete time instances Δ, 2Δ, 3Δ .... the system is called discrete time. Where,

Δ = Unit delay

t = 0 , is the initial state

for a discrete – time Artificial Neural System,

$$\underline{O}^{k+1}=\Gamma\left[\underline{W}\;\underline{O}^{k}\right] \qquad \text{for k = 1, 2 ....} \qquad (1.11a)$$

$$\underline{O}^1 = \Gamma\left[\underline{W}\ \underline{x}^o\right]$$

$$\underline{O}^2 = \Gamma\left[\underline{W}\ \Gamma\ \underline{w}\ x^0\right]$$

$$\underline{O}^{k+1} = \Gamma\left[\underline{W}\ \Gamma\ [\ldots\ldots\ldots\ldots\Gamma\ [\underline{w}\ \underline{x}^o]]\right] \tag{1.11b}$$

The above network is called recurrent since its response at k+1th instant depends on the entire history of the network starting at k=0. Recurrent networks operate with discrete representation of data; they employ neurons with hard-limiting activation function. However, if we have infinitesimal delay then the output vector may be assumed to be continuous time function.

1.3 Knowledge Representation

The primary characteristics of knowledge representation are twofold: (1) what information is actually made explicit; and (2) how the information is physically encoded for subsequent use. A major task for a neural network is to learn a model of the world in which it is embedded and to maintain the model sufficiently consistent with the real world so as to achieve the specified goals of the application of interest. In a neural network, the subject of knowledge representation is very complex. Since there are multiple sources of information that activate the network, and these sources interact among themselves, adding to the complication. However, there are four rules for knowledge representation that are of general common-sense nature. These four rules are:

Rule1. Similar inputs from similar classes should generally produce similar representation inside the network, and should therefore be classified as belonging to the same category.

Rule 2. Items to be categorized as separate classes should be given widely different representation in the network.

Rule 3. If a particular feature is important, then there should be large number of neurons involved in the representation of that item in the network.

Rule 4. Prior information and invariance should be built into the design of a neural network, thereby simplifying the network design by not having to learn them.

1.4 Artificial Intelligence and Neural Network

The aim of *artificial intelligence* (AI) is the development of algorithms that require *cognition* when performed by humans. This statement on AI is adapted from Sage(1990). An AI system must be capable of doing three things: (1) store knowledge; (2) apply the knowledge stored to solve problems; and (3) acquire new knowledge through experience. An AI system has three key components: representation, reasoning and learning.

1.4.1 **Representation**: AI uses the language of *symbols* structure to represent both general knowledge about a problem of interest and specific knowledge about the solution to the problem. The symbols are formulated in familiar terms making symbolic representations of AI relatively easy to understand by a human user. "Knowledge," in the case of AI is basically data. It may be declarative or procedural kind. In a *declarative* representation, knowledge is a static collection of facts, with a set of procedures used to manipulate these facts. In the *procedural* representation, knowledge is enclosed in an executable code that acts out the meaning of the knowledge. Both the kinds of representation are needed in most of the problems.

1.4.2 **Reasoning**: It's the ability to solve problems. A system must satisfy certain conditions to be classified as a reasoning system (Fischler and Firschein, 1987):

- The system must be able to express and solve a wide range of problems.
- The system should be able to make *explicit* any *implicit* information available to it.

- The system must have a control mechanism that indicates which operation should be applied to a particular problem, when a solution to the problem has been found, or when further work on the problem should be terminated.

Problem solving can be thought as a *searching* problem. Search is carried out using *rules, data,* and *control.* The rules operate on the data and the control operates on the rules (Nilsson, 1980). In many problems the available knowledge is either incomplete or it can not be extracted. In such cases, *probabilistic reasoning* procedures are used, thereby permitting AI systems to take the uncertainty of the problem in account.

1.4.3 **Learning**: As far as machine learning is concerned we can broadly classify it into three types.

i. Inductive

ii. Deductive

iii. Abduction

These are three ways in which the available information may be processed. Inductive processing deals with determining general patterns, or organizational schemes rules and laws from raw data, experience or examples. Inductive computations perform abstractions, producing generalities from specifics. The creation of models and theories is basically inductive work. Whereas determining of specific facts using general rules is deductive information processing. Also determination of new general rules from old ones is called deductive for examples a general rule can be area of a circle is "pi times square of radians", from this we deduce that a circle of radii 5 cm. Has an area of ≈(π-25). Proof of theorems is also a deductive process as it requires rules of inference and previous theorems. When learning is concerned with enlargement of knowledge base, then one generally asks these questions. "Is the new knowledge put in directly", "Is it induced from examples or experience" or "Is it deduced from existing knowledge?"

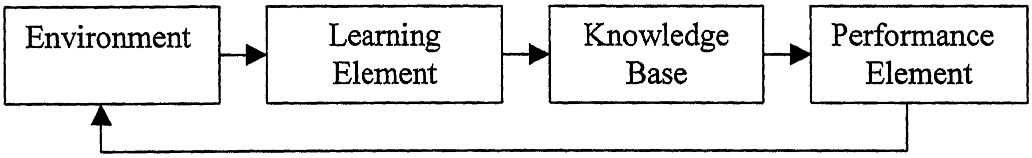


Figure 1.5: Simple model of machine learning

1.5 AI versus Neural Network

The comparison between these two may be carried out under three sub-divisions as mentioned below:

(i) Level of Explanation

(ii) Style of Processing

(iii) Representational Structure

1.5.1 **Level of Explanation**: In the case of AI the emphasis is on the symbolic representation. The representations are discrete and arbitrary: abstract properties, and not analog images. AI assumes existence of mental representations, and it models cognition as the sequential processing of symbolic representations (Newell and Simon, 1972). In the case of the neural network the assumption made regarding the cognitive process is entirely different from those in classical AI. The emphasis in neural networks is on the development of *parallel distributed processing* (PDP) *models*. These models assume that information processing takes place through interaction of a large number of neurons, each of which sends excitatory and inhibitory signals to other neurons in the network (Rumelhart and McClelland, 1986).

1.5.2 **Processing Style**: In the classical AI, the processing is sequential. The operations are performed in a step-by-step manner. Whereas, neural networks employ parallel processing. This is a source of flexibility and robustness to the neural network. In the case of neural network the computation is spread over a large number of neurons. This is the reason why it can recognize noisy or incomplete inputs also. A damaged network may still be able to function satisfactorily, and the learning does not have to be perfect. Thus, parallel distributed processing approximates the flexibility of a continuous system, in sharp contrast with the rigid and brittle discrete symbolic AI.

1.5.3 **Representational Structure**: With a language of thought pursued as a model for classical AI, symbolic representations possess a *quasi-linguistic structure.* The expressions of classical AI are generally complex, built in a systematic fashion from simple symbols. Given a limited stock of symbols, new expressions are composed by the virtue of *compositionality* of symbolic expressions and the analogy between syntactic structure and semantics.

The nature and structure of representations, in the case of neural network, is a crucial problem. In a neural network, representations are distributed. It does not, however, follow that whatever is distributed must have constituents, and being distributed is very different from having semantic or syntactic constituent structure (Fodor and Pylyshyn, 1988). Unfortunately, most of the neural network models proposed to date for distributed structure representations are rather *ad hoc*; they solve the problem for a particular class in a way that cannot be extended easily.

To sum up, symbolic AI may be described as a formal manipulation of a language of algorithms and data representations in a *top-down* fashion. On the other hand, neural networks may be described as parallel distributed processors with a natural learning capability, and which usually operates in *bottom-up* fashion. Thus a better approach of implementing a cognitive task would be one that combines both these methods.

1.6 Objective

The objective of this Thesis is to study the performance of Artificial Neural Networks designed using MATLAB Toolbox on typical and general benchmarking problems, and comparing these to the performance of Networks designed using codes written in JAVA language. The problems considered cover most of the aspects of neural network theory. The problems considered belong to these two categories: Classification and Function Approximation. This thesis has been organized in five chapters, these are:

Chapter 1: Introduction and Problem Definition

Chapter 2: Feed-forward ANN Models

Chapter 3: The Back-propagation Algorithm

Chapter 4: Learning

Chapter 5: Benchmark Problems

Chapter 6 : Results and Discussions

Chapter 2

Feedforward ANN Models

Most methods of training multi-layered Artificial Neural Networks (ANN) are based on the steepest descent technique, which frequently have problems such as very poor convergence behavior, trapping in local minima, misdirection of descent, and oscillation. Furthermore the ANN suffers from problems like network paralysis, overgeneralization, and multiple solution problems. Therefore the objective of the chapter is to discuss the methodologies to overcome the above problems. Development of ANN is performed in two phases:

1. Training phase: In this phase the ANN tries to memorize the pattern of learning data set. This phase consists of the following modules:

- Selection of Neuron Characteristics
- Selection of Topology
- Error Minimization Process
- Selection of Training Pattern and Preprocessing
- Stopping Criteria of Training

2. Testing phase: Here ANN tries to predict and test data sets.

2.1 Training Phase Issues

2.1.1 Selection of Neuron Characteristics

Neurons can be characterized by two operations: Aggregation and Activation function. Non-linear filtering (also called activation function) can be characterized by several functions: sigmoidal and tangent hyperbolic functions being the most common. Both have similar transfer characteristics and hence mapping properties. Various other activation functions are linear, signum (thresholding function), logarithmic, sinusoidal etc. Even though numerous activation functions have been reported in literature, care should be exercised in selecting the activation function. Selection of the function is problem dependent. For example, the logarithmic activation is used when the upper limit is unbounded while the Radial Basis function (RBF) is used for problems having complex boundaries. The main objective is to design an ANN of superior generalization ability and lesser number of nodes. This avoids unnecessary calculations and it can be used for fast decision-making.

Certain class of problems can be handled more suitably by fuzzy networks. In fuzzy networks some nodes are made up of membership function, π and S type. The selection of membership function is again problem dependent and can be handled accordingly.

Since most of the ANN use the gradient method to minimize error, points where the gradients are small should be avoided in the beginning of the process(small gradients make the learning process slow). Therefore initialize the learning process such that the central part of the function where the gradients are large is selected. This can be achieved by proper scaling of input signals and the selection of limit between which random weights are generated. For example, if the scaling of input is carried out between 0.1 and 0.9 then the random weights are preferred between –0.5 and +0.5 for standard sigmoidal function. A major drawback of these activation functions is their saturation, which can cause significant errors when the output has no upper bound.

2.1.2 Selection of Topology

Topology of ANN deals with *the number of neurons in each layer* and *their interconnections.* Too few hidden neurons hinder the learning process, while too many occasionally degrade the ANN generalization capability. There are no clear-cut or absolute guidelines available in the literature for deciding the topology of ANN. However numerous researchers have evolved their own thumb rules and heuristics.

One rough guideline for choosing the number of hidden neurons is the *geometric pyramid rule* . It states that, for many practical networks, the number of neurons follow, in successive layers, pyramidal shape, decreasing from input towards the output. The optimal number of hidden neurons can be determined by using **rigorous search**. The number of neurons is fixed in the input and the output layers and therefore, in the starting the ANN is trained and tested with the least number of hidden neurons. If the net is not being trained then the number of neurons in the hidden layers is increased gradually till the error is acceptable. The **rigorous search** method, as shown in the figure 2.1, is time consuming but reliable.

Pruning of ANN can reduce time complexity of topology decision. Starting with a large ANN architecture does this. Large architecture gives less generalization. Later, some of the parts of the network are pruned to improve generalization and sensitivity. There are various algorithms available in literature for this.

2.1.3 Error Minimization Procedures

When ANN are trained, the weights of the links are changed/adjusted so as to achieve minimum error. During this process a part of the entire data set is used as training set, and the error is minimized on this set. Calculations for error may be done using various functions. The most commonly used being the *Root Mean Square* (**RMS**) function and is given by:

$$\text{Error} = (1/2)^* \sqrt{} (\Sigma[(\text{actual} - \text{predicted})^2]) \qquad (2.1)$$

Equation 2.1 can be generalized as:

$$\text{Error} = (1/2)^* \sqrt{} (\Sigma[(\text{actual} - \text{predicted})^k]) \qquad (2.2)$$

Where k = Order of Norm

We can also use an error function such as

$$\text{Error} = \Sigma\text{Actual}^*\log(\text{predicted}) + (1 - \text{Actual}) \log(1 - \text{Predicted}) \qquad (2.3)$$

This error function may increase the learning rate of ANN and its generalization capability.

Generally techniques based on calculus are used to minimize error. These methods have very poor convergence, as they may get trapped in local minima, a drawback, which can be eliminated by using evolutionary techniques such as Simulated Annealing and Genetic Algorithms. Unfortunately these techniques are not applicable for fast training.

Different strategies are used for updating the weights of ANN. If the weights are updated after presentation of the entire set of training patterns to the network, then it is called *batch training.* In the second approach, the weights are updated after presentation of each training pattern, this is called *pattern training or online updating* . In another approach, frequency of the given training samples for updating the weights are also dependent on its error.

In fact learning process may be split into two groups, Active and Passive. In Passive Learning, all available examples are fed to the ANN for learning, whereas in the case of Active learning it is not so. In the case of Active learning ANN is trained with pattern corresponding to maximum error. After the network has been trained, it is tested on the remaining examples. The samples giving the worst error are then added to the training set. This process is repeated until the ANN performs well on the remaining data set. Thus the cross validation is done automatically, in this case.

Training of ANN is influenced by selection of different parameters. Learning and momentum rate influence the speed of learning in the gradient decent techniques. Increasing the learning rate may improve the speed of training but it may also lead to oscillations, whereas a low learning rate may cause slow learning but it will be stable. Scaling parameter (gain), in the case of sigmoidal function, also affects the learning behavior.

2.1.4 **Selection of Training Pattern and Preprocessing**

Selection of the training data is very critical for building ANN. Although preprocessing is not mandatory, but it definitely improves the performance of the network and reduces the learning time. Some commonly used preprocessing methods are: Scaling, Normalization, and Noise reduction. In the case of pattern recognition problems, noise is purposely added (preprocessing) to improve the recognition capability of the network.

Several other transformations like linear and log scaling of training data is required for good prediction. If the distribution of variables is unusual, it may be more difficult for the ANN to learn to use it, even if the variable is linearly scaled to a reasonable range. This is because the information content in the variable is too distorted. Small but important variations of the variable may be compressed into a relatively narrow area, while other variations may be spread out in a wider range than it is required. Such type of data set requires non-linear scaling. Examination of distribution of variables before and after the transformation must be done. Preprocessing definitely gives better results and is recommended to be done as a routine.

2.1.5 Termination Criteria for Training

The process of adjusting weights of an ANN is repeated until the termination condition is met. The training process may be terminated under if any one of these conditions is met:

- The error goes below a specific value
- The magnitude of gradient reaches below a certain value
- Specific number of iterations is complete

The network training can be terminated by cross validation technique also, as shown in figure 2.2. In this technique the data set is divided into a training set and a validation set. The training set is used to modify the connection weights and the validation set is used to estimate the generalization ability. Training is stopped when the error on the validation set begins to rise. This method may not be practical for a small data set.

The first three criteria are sensitive to the choice of parameter and may lead to poor results if the parameters are not properly selected. The cross validation does not have this drawback. It can also avoid over fitting of the data as shown in the figure 2.4 and it actually improves the generalization performance of the network. However this technique is time insensitive.

2.2 Testing Phase Issues

The performance of ANN on the testing data set represents its generalization ability. In actual practice the data set is divided into two. One set is used for training and the other for testing. In fact a generalized neural network will perform well for both training and testing data. Testing must be done for interpolation as well for extrapolation. Although it has been found that the performance of ANN for extrapolation is generally poor. During the process of testing some of the weights of ANN are removed to check the stability. If the performance of ANN does not decline then it implies that the ANN will be stable in case of small hardware/software failure. Testing of ANN only with undistorted testing data is not sufficient, a small amount of noise must be added to the data to check the stability of the network.

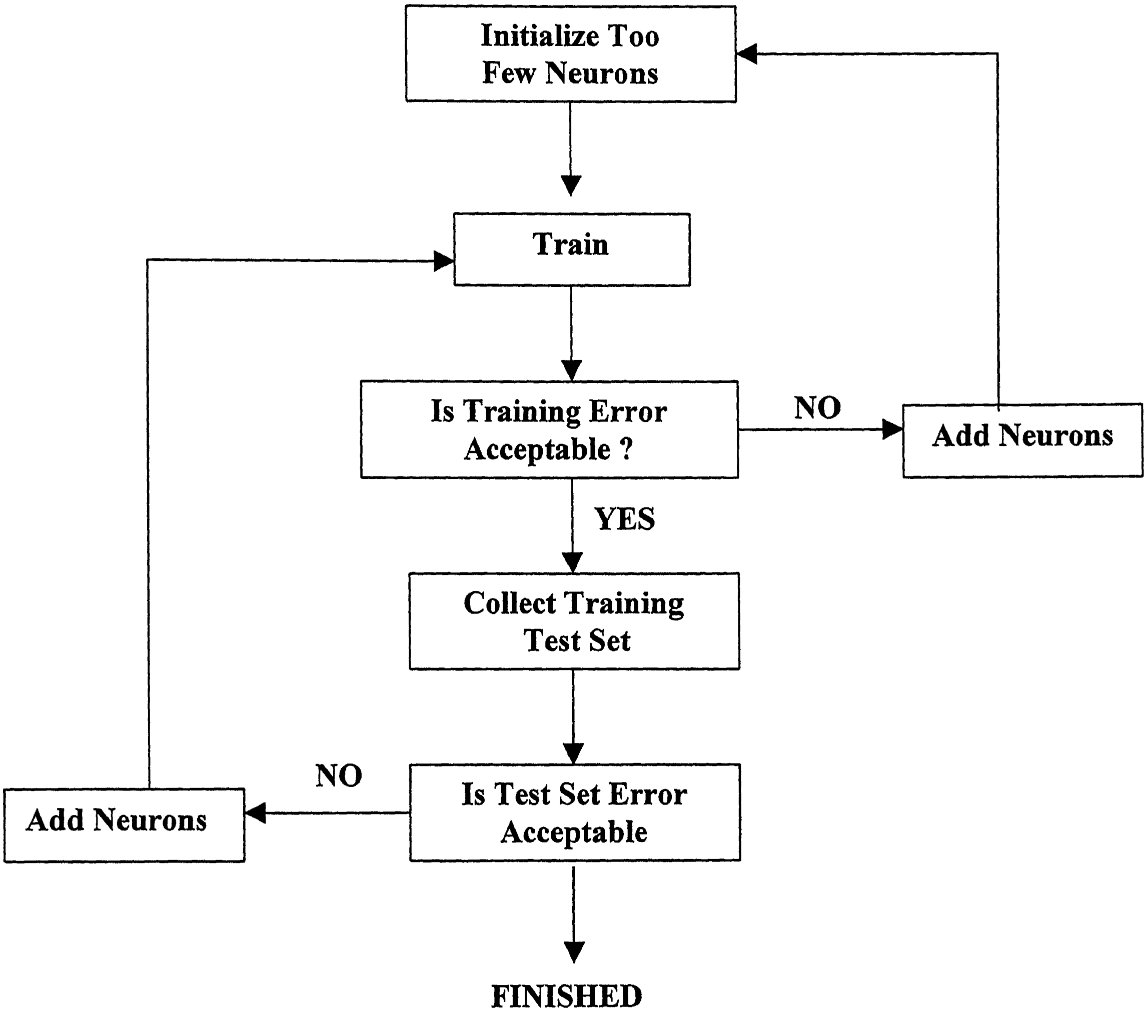

Initialize Too Few Neurons
Train
Is Training Error Acceptable ?
NO
Add Neurons
YES
Collect Training Test Set
NO
Add Neurons
Is Test Set Error Acceptable
FINISHED

Figure 2.1: Method for Deciding the Topology of ANN

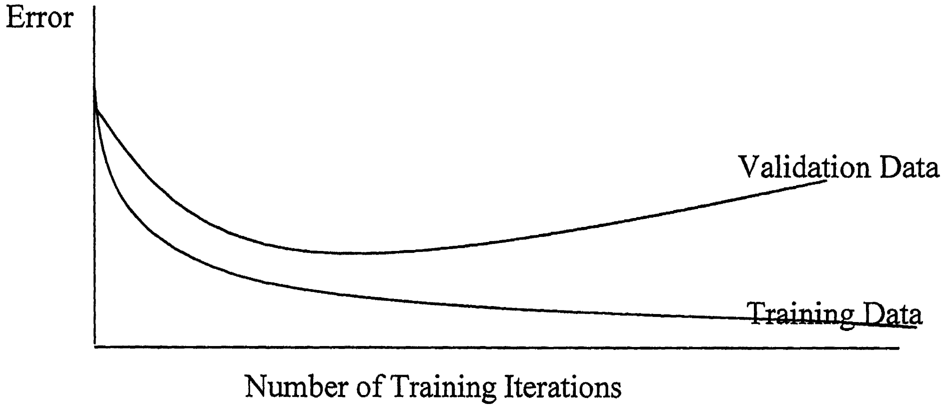


Figure 2.2 : An Example of Validation of Data

Chapter 3

The Backpropagation Algorithm

The Error-Back-propagation or Multi-layer Feed Forward or Backprop, as shown in figure 3.1, are used to represent the Back-propagation algorithm. It is the most important algorithm for the supervised learning of multi-layer feed forward ANNs. Here the error signals are propagated backwards through the network on a layer-by-layer basis.

The backprop algorithm is based on the selection of a suitable error function or cost function, whose values are determined by the actual and the desired outputs of the network and which is also dependent on the network parameters such as weights and thresholds. The basic idea is that the cost function has a particular surface over the weight space and therefore an iterative process such as the gradient descent method can be used for its minimization. The method of gradient descent is based on the fact that since the gradient of a function always points in the direction of maximum increase of the function, thus moving in the direction of negative gradient induces a maximal "downhill" movement that will eventually reach the minimum of the function surface over its parameter space. This is a rigorous and well established technique for minimization of functions and has probably been the main factor behind the success of back-propagation. However, this method does not guarantee that it will always converge to minimum of error surface as the network can be trapped in various local minima.

The backprop algorithm training consists of two passes of computations: a *forward pass* and a *backward pass,* as shown in figure 3.2. During the forward pass the weights are fixed whereas during the backward pass, the synaptic weights are adjusted in accordance with *error correction rule (error = desired – actual).* In the forward pass an input pattern vector is applied to the input layer nodes. The signal from the input layer propagates to the units in the first layer and each unit produces an output according to the equation:

$$Y_j = 1/(1 + \exp(-V_j)$$

V_j = Net internal activity level of neuron j

Y_j = Output of neuron = $f(\sum_{i=0}^{i=N} w_i x_i \ )$

The outputs of these units are propagated to the units in the subsequent layers and the process goes on until the signals reach the output layer where the actual response of the network to the input vector is obtained. In the backward pass the synaptic weights are adjusted in accordance with an error signal, which is propagated backwards through the network against the direction of synaptic connection.

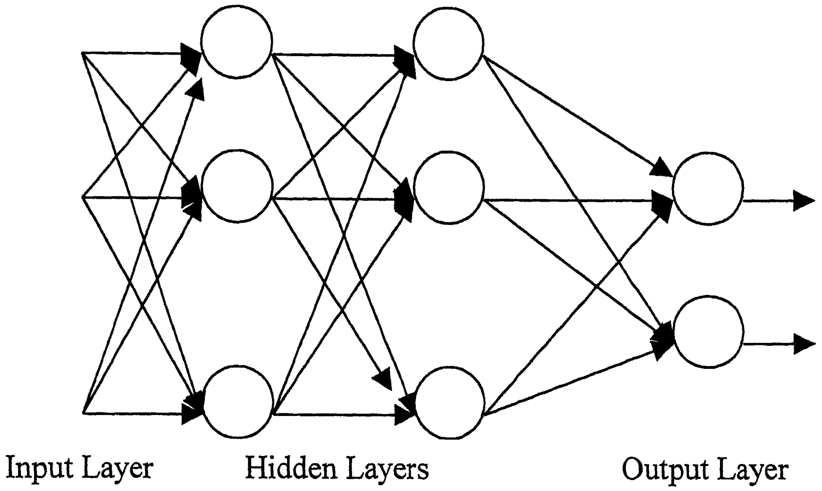


Figure 3.1: A feed-forward network

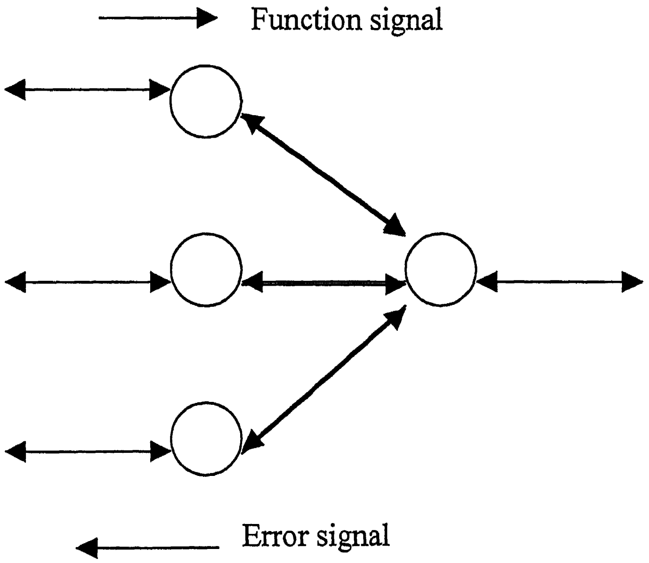


Figure 3.2 : Flow of function and error signals

3.1 Mathematical Analysis of Backpropagation Algorithm

Forward pass : Given an input pattern vector $Y^{(p)}$, each hidden node j receives a net input

$$x_j^{(p)} = \sum_k w_{jk} y_k^{(p)} \tag{3.1}$$

w_{jk} = weight between hidden node j and input node k

Output of node j is given by

$$y_j^{(p)} = f(x_j^{(p)}) = f(\sum_k w_{jk} y_k^{(p)}) \tag{3.2}$$

Input to each output node i is given by

$$x_i^{(p)} = \sum_j w_{ij} y_j^{(p)} = \sum_j w_{ij}\ f(\sum_k w_{jk} y_k^{(p)}) \tag{3.3}$$

w_{ij} = weight between output node i and hidden node j.

Final output from the output node i

$$y_i^{(p)} = f(x_i^{(p)}) = f(\sum_{kj} w_{ij} y_j^{(p)}) = f(\sum_j w_{ij}\ f(\sum_k w_{jk} y_k^{(p)})) \tag{3.4}$$

Backprop algorithm can be implemented in two different modes: *on line mode* and *batch mode*. These are discussed in brief.

Online mode: The error function is calculated after the presentation of each input pattern and the error signal is propagated back through the network modifying the weights before the next input pattern is applied. The most commonly used error function is the Mean Square Error (MSE) of the difference between desired and the actual responses of the network over all the output units. After this the new weights remain fixed and a new pattern is presented to the network and this process continues until all the patterns have been presented to the network. Presentation of the entire set of patterns is termed as one epoch or one iteration.

Batch mode : The error signal is calculated for each input pattern but the weights are modified only when all input patterns have been presented. The error function is calculated as the sum of the individual MSE errors for each pattern and the weights are accordingly modified (all in a single step for all the patterns) before the next iteration. Error function in batch mode, calculated as MSE over all output units i and over all patterns p is given by:

$$\begin{aligned} \mathrm{E} &= \frac{1}{2}\sum_{p}\sum_{i}\left(d_i^{(p)} - y_i^{(p)}\right)^2 \\ &= \frac{1}{2}\sum_{p}\sum_{i}\left(d_i^{(p)} - f\left(\sum_{j} w_{ij} f\left(\sum w_{jk} y_k^{(p)}\right)\right)\right)^2 \end{aligned} \tag{3.5}$$

E here is differentiable function of all the weights (and thresholds according to the bias convention) and therefore we can apply the method of gradient descent

For the hidden-to-output connections the gradient descent rule gives

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} \tag{3.6}$$

Where η is a constant, it determines the rate of learning, and is therefore called the learning rate of the backpropagation algorithm. Using chain rule on the above equation we have

$$\left.\begin{aligned} &\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial y_i^{(p)}} \frac{\partial y_i^{(p)}}{\partial W_{ij}} \\ \text{and} \\ &\frac{\partial y_i^{(p)}}{\partial W_{ij}} = \frac{\partial y_i^{(p)}}{\partial x_i^{(p)}} \frac{\partial x_i^{(p)}}{\partial W_{ij}} \end{aligned}\right\} \tag{3.7}$$

from these two equations we have

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial y_i^{(p)}} \frac{\partial y_i^{(p)}}{\partial x_i^{(p)}} \frac{\partial x_i^{(p)}}{\partial W_{ij}} = -\sum_{p}\left(d_i^{(p)} - y_i^{(p)}\right) f'\left(x_i^{(p)}\right) y_j^{(p)} \tag{3.8}$$

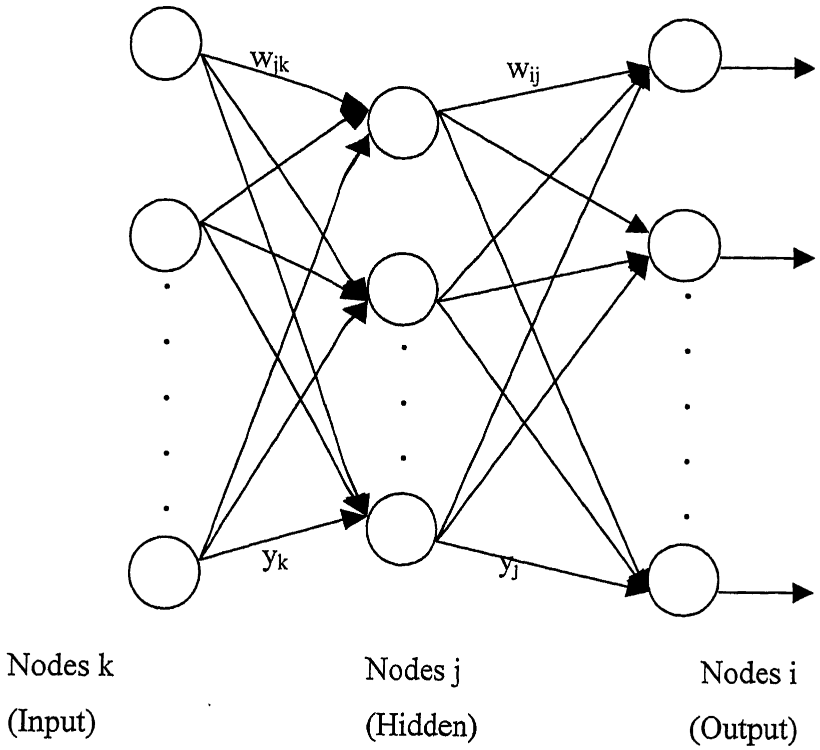


Figure 3.3: Multi-layer Feed-forward Network

Thus,

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} = \eta \sum_{p} \left(d_i^{(p)} - y_i^{(p)}\right) f'(x_i^{(p)}).y_j^{(p)}$$

$$\Delta W_{ij} = -\eta \sum_{p} \delta_i^{(p)} y_j^{(p)} \qquad (3.9)$$

Where $\delta_i^{(p)} = (d_i^{(p)} - y_i^{(p)})\, f'\,(x_i^{(p)})$ (3.10)

For Input-to-hidden connections the gradient descent rule gives the weights by the following equations:

$$\Delta W_{jk} = -\eta \frac{\partial E}{\partial W_{jk}} \qquad (3.11)$$

Applying the chain rule to the above equation we get

$$\left.\begin{aligned} &\frac{\partial E}{\partial W_{jk}} = \frac{\partial E}{\partial y_j^{(p)}} \frac{\partial y_j^{(p)}}{\partial W_{jk}} \\ &\text{and} \\ &\frac{\partial y_J^{(p)}}{\partial W_{Jk}} = \frac{\partial y_j^{(p)}}{\partial x_j^{(p)}} \frac{\partial x_j^{(p)}}{\partial W_{jk}} \end{aligned}\right\} \qquad (3.12)$$

From both these equation we get

$$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial y_i^{(p)}} \frac{\partial y_i^{(p)}}{\partial x_j} \frac{\partial x_j^{(p)}}{\partial W_{jk}} \quad = \quad \frac{\partial E}{\partial y_j^{(p)}} f'\left(x_J^{(p)}\right) y_k^{(p)} \qquad (3.13)$$

The term $\frac{\partial E}{\partial y_j^{(p)}}$ can be represented as

$$\begin{aligned} \frac{\partial E}{\partial y_j^{(p)}} &= - \sum_p \sum_i \left(d_i^{(p)} - y_i^{(p)}\right) \frac{\partial f\left(x_i^{(p)}\right)}{\partial y_j^{(p)}} \\ &= - \sum_p \sum_i \left(d_i^{(p)} - y_i^{(p)}\right) \frac{\partial f\left(x_i^{(p)}\right)}{\partial x_i^{(p)}} \frac{\partial x_i^{(p)}}{\partial y_i^{(p)}} \\ &= - \sum_p \sum_i \left(d_i^{(p)} - y_i^{(p)}\right) f'\left(x_i^{(p)}\right) W_{ij} \end{aligned} \qquad (3.14)$$

Thus ,

$$\frac{\partial E}{\partial W_{ij}} = \sum_p \sum_i \left(d_i^{(p)} - y_i^{(p)}\right) f'\left(x_i^{(p)}\right) W_{ij} \; f'\left(x_j^{(p)}\right) y_k^{(p)} \qquad (3.15)$$

$$\begin{aligned} \Delta W_{jk} = -\eta \frac{\partial E}{\partial W_{jk}} &= \eta \sum_p \sum_i \left(d_i^{(p)} - y_i^{(p)}\right) f'\left(x_i^{(p)}\right) W_{ij} \; f'\left(x_j^{(p)}\right) y_k^{(p)} \\ &= \eta \sum_p \sum_i \delta_i^{(p)} \; f'\left(x_j^{(p)}\right) W_{ij} \; y_k^{(p)} \end{aligned}$$

Therefore

$$\Delta W_{jk} = \eta \sum_p \delta_j^{(p)} y_k^{(p)} \qquad (3.16)$$

Where

$$\delta_j^{(p)} = f'\left(x_j^{(p)}\right) \sum_i \delta_i^{(p)} W_{ij} \qquad (3.17)$$

For equations 3.8 and 3.15 we see that if the activation function f(.) was not differentiable then we would be unable to implement the gradient descent rule since it would be impossible to calculate the partial derivatives of E with respect to the weights. THAT IS WHY DIFFERENTIABILITY of ACTIVATION FUNCTION is so important in the back-propagation learning. Consider sigmoid function, its derivative is given by

$$f'(u) = f(u)\,[1 - f(u)] \qquad (3.18)$$

This shows that we need not compute f'(u) once we know f(u).

Where,

$f(u) = 1/(1 + \exp(-au))$ is an example of *sigmoid function* (Logistic function). Equation 3.18 highlights the importance of using sigmoid function as an activation function. The activation time is reduced significantly as it is not necessary to calculate f'(u) separately.

On-Line Mode of training: In this, the weight updates are given by

$$\Delta W_{ij} = \eta\ (d_i - y_i)\ f'(x_i)\ y_j = \eta\ \delta_i\ y_j \qquad (3.19)$$

$$= \eta\ \delta_i\ y_j$$

$$\Delta W_{jk} = \eta\ \delta_j\ y_k \qquad (3.20)$$

The difference, as compared to batch learning is that, here there is no summation over the set of training patterns. The two equations are the same except with a different definition of δ. In general, with an arbitrary number of layers, the back-propagation update rule always has the form:

Weight Correction = (Δ w_{lm}) = Learning Rate(η)* Local Gradient (δ_l)* Input signal(y_m)

This is called the *Generalized Delta Rule.*

3.1.1 **Initialization**: An important aspect of back-propagation training is the proper initialization of the network. Improper initialization may cause the network to require a very long time for training and there is a high probability that the solution eventually found may not be the optimum solution. The first step in the back-propagation algorithm is initialization of the network. A good choice of initial values of the free parameters (i.e. weights and thresholds) of the network can significantly accelerate learning. If all the weights start off with equal values and if the solution requires that unequal weights be developed, the system can never learn.

3.1.2 **Learning Rate** : Another important parameter is the learning rate. It determines what amount of the calculated error sensitivity to weight change will be used for weight correction. The "best" value of learning rate depends on the characteristics of the error surface, i.e., a plot of E versus w_{ij}. If the surface changes rapidly, the gradient calculated only on local information will give poor indication of the true "right path." In such a case smaller learning rate is desirable. On the other hand, if the surface is relatively smooth, a large learning rate will speed up convergence. This logic however requires a prior knowledge of the error surface, which is rarely available. A general rule is to use the largest learning rate that works and does not cause oscillation. A rate too large may cause the system to oscillate and thereby slow or prevent the network's convergence.

3.1.3 **Momentum Method:** The purpose of the momentum method is to accelerate the convergence of the error back-propagation learning algorithm. In this method the weight update is supplemented with a fraction of the most recent weight adjustment. This is a very simple method of increasing the learning rate and yet avoids the danger of instability. The weight update is done according to the formula:

$$\Delta W_{jk}(n) = -\eta \frac{\partial E}{\partial W_{jk}} + \alpha\ \Delta W_{jk}(n-1) \tag{3.21}$$

where the arguments *n* and *n-1* are used to indicate the current and the most recent training steps, respectively, and α is a user selected positive momentum

3.2 Activation Functions

Neural networking theory shows that backprop networks can represent most reasonable functions as close as you like with linear output units and a single layer of non-polynomial hidden layer units. There are however many activation functions that you can choose from and each one has its own special virtues.

The original network activation function was the *linear activation function*:

$$y = D*x \qquad (3.22)$$

where x is the input to the neuron, y is the final value of the neuron and usually D = 1. This is also called identity mapping. The figure below shows some linear activation functions

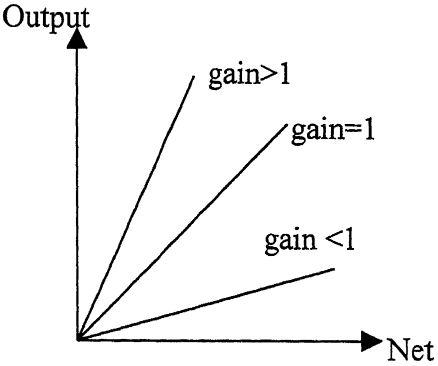


Figure 3.4 : Linear Activation Functions

Linear functions are inadequate to approximate most of the functions, and therefore, some non-linear functions are needed.

- The *standard sigmoid* (or logistic) runs from 0 to 1 and it is:

$$y = 1/(1+\exp(-D*x)) \qquad (3.23)$$

where the input to the neuron is x and most often D=1. The derivative is: y * (1 – y). There is some theory and a few experiments that show that hidden layer unit activation values centered around 0 will speed up training so in some cases people subtract 0.5 from the above. The standard sigmoid can be approximated using the function

x	f(x)
x >= 1	1
-1 < x < 1	0.5 + x * (1 – abs(x) / 2)
x <= -1	0

(3.24)

if you use x = input / 4.1. The maximal absolute deviation between this function and the standard sigmoid is less than 0.02. The figure 3.5 shows some of the logistic activation functions used:

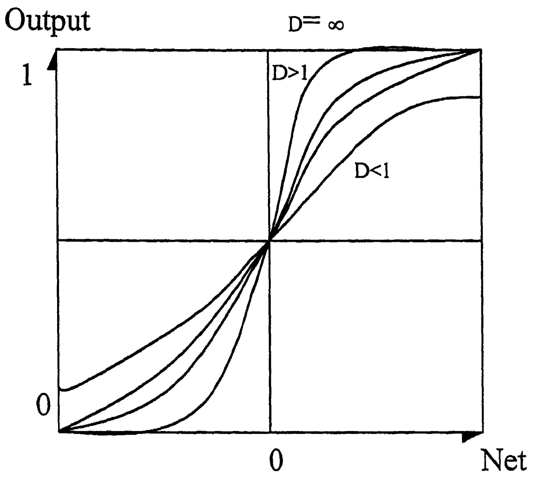


Figure 3.5: Logistic Activation Functions

- Another popular function is *tanh*, it has outputs in the range –1 to 1 and it can be written as:

$$y = 2 / (1 + \exp(-2 * x)) - 1 \tag{3.25}$$

The derivative is: 1 – y * y. Because its values are centered around 0 there is no reason to believe that using tanh will result in faster training. Experiments show that sometimes tanh is better but sometimes it is not.

- One can also use approximation of sigmoid functions with a series of straight lines, a piecewise linear function. These may require more iteration to solve the problem but even so it will save CPU time.

- Sometimes the *Gaussian function*, (as shown in figure 3.6):

$$y = \exp(- x * x) \tag{3.26}$$

is used and in some cases it can produce faster training, for instance in the 2-1-1 XOR network with direct input to output connections you can get faster training. The Gaussian also improves the performance of the 10-5-10 ENCODER problem. The derivative is: -2*x*y, where x is the input and y is the value of the neuron.

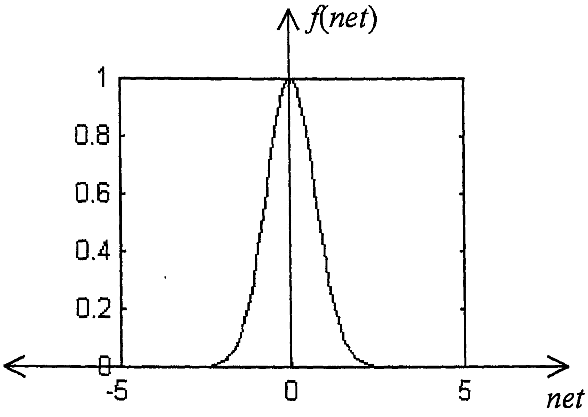


Figure 3.6: Gaussian Activation Function

- The following sigmoid runs from –1 to 1 and is also faster to compute

y = x/(1 + |x|) (3.27)

The derivative is : 1/((1 + |x|)*(1 + |x|))

- This sigmoid runs from 0 to 1 and is also faster to compute:

y = (x/2) / (1 + |x|) + 0.5 (3.28)

Its derivative is given by : 1 / (2*(1 + |x|)*(1 + |x|)

The last two sigmoids approach their extremes more slowly. This means that if you are trying to output numerical values it will take more iterations to reach your target value. But if you are doing a classification problem you really care to get the correct output value greater that other outputs and here these functions will save time without influencing the number of iterations required by very much.

Theory says that backprop can approximate most normal functions if and only if the hidden later unit is non-polynomial however the following function can also work:

y = sgn(x) * x * x (3.29)

and it has the virtues of running from minus infinity to plus infinity and being fast to compute. The derivative is sgn(x)*2*x. However, quite often the calculations will go wild unless very small learning rates are used, or better still use an acceleration algorithm that will automatically control the learning rates.

3.3 Faster Training

Plain back-propagation is terribly slow and everyone wants it to go faster. There are a series of things that can be done to speed up learning:

- Fudge the derivative term
- Scale the Data
- Direct Input-Output Connections
- Vary the Sharpness (Gain) of the Activation Function
- Use a different Activation Function
- Use better Algorithms

3.3.1 **Fudge the Derivative Term:** The first major improvement to backprop is extremely simple: you can fudge the derivative term in the output layer. If using the usual backprop activation function:

$$1 / (1 + \exp(-D*x))$$

the derivative is

$$s * (1 - s)$$

where s is the activation value of the output unit and most often D =1. The derivative is largest at s = ½ and it is here that you will get largest weight changes. Unfortunately, as 0 or 1is approached, the derivative term gets close to 0 and the weight changes become very small. In fact if the network's response is 1 and the target is 0, that is the network is off by quite a lot, you end up with very small weight changes. It can take VERY long time for the training process to correct this. More than likely one gets tired of waiting. Fahlman's solution was to add 0.1 to the derivative term making it :

$$0.1 + s*(1 - s)$$

The solution of Chen and Mars was to drop the derivative term altogether, in effect the derivative was 1. This method passes back much larger error quotas to the lower layer, so large that a small *eta* must be used there. In their experiments on the 10-5-10 encoder problem they found the best results came when that *eta* was 0.1 times the output level *eta*, hence they called their method the "differential step size" method. One tenth is not always the best value so one must experiment with both the upper and the lower level etas to get the best results. Besides that, the *eta* used for the upper layer must also be much smaller than the *eta* used without this method.

3.3.2 **Direct Input-Output Connections:** Adding direct connections from the input layer to the output layer can often speed up training. It is supposed to work best when the function to be approximated is almost linear and it only needs a small amount of adjustment from nonlinear hidden layer units. This method can also cut down on the number of hidden layer units you need. It is not recommended when there are a large number of output units because then you add more free parameters to the net and possibly hurt generalization.

3.3.3 **Adjusting the Sharpness/Gain**: The training time can be decreased by increasing the sharpness or gain (D) in the standard sigmoid:

$$1/\left(1 + \exp(-D*x)\right)$$

In fact they show that the training time goes as 1/D for training without momentum and 1/sqrt(D) for networks with momentum. This is not a perfect speed-up scheme since when D is too large you run the risk of becoming trapped in a local minimum. Sometimes the best value for D is less than 1. This can be used in combination with all other training algorithms.

3.3.4 **Better Algorithms:** It is desirable to have faster training and there are many variations on backprop that will speed up training times enormously. However, from experience it has been found that very slow online update method will sometimes give better results as compared to these faster algorithms, although these may be pitifully slow. In most cases the accelerating algorithms work so much faster than either online or batch backprop, so one must first try these faster methods and then if better results are desired then try either the slower online update or batch method. There is a whole collection of algorithms where different learning rates (eta) are assigned to each weight. As the training proceeds eta is increased in some way if the error is reducing (downhill).

When the weight change becomes too large then the net lands on the hill and then the learning rate is required to be reduced. Algorithms vary on both the speeding up and slowing down details. Second there is a set of algorithms known as conjugate gradient methods. Third there are methods that build up the network one hidden unit at a time. These are some of common algorithm, most of these methods involve extra calculations to determine the value of the learning rate and momentum term.:

- The Rprop Algorithm
- The Quikprop Algorithm
- The SuperSAB Algorithm
- Conjugate Gradient
- Cascade Correlation
- Variable Learning Rate Algorithms(GRBH, IG)

The Rprop Algorithm : Backprop uses sigmoid transfer functions in its hidden layers. These functions are called squashing functions, since they compress an infinite input range into a finite output range. Their slope approaches zero as the input gets large. This causes problem when using steepest descent to train a multi-layer network with sigmoid functions, since the gradient can have a very small magnitude, and therefore cause small changes in the weights and the biases, even though the weights and the biases are far from their optimal values. The purpose of resilient backprop training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight update is determined by a separate update value. The update value for each weight and bias is increased by a factor delt_inc whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. The update value is decreased by a factor delt_dec whenever the sign changes with respect to the previous iteration. If the derivative is zero, then the update value remains the same. Whenever the weights are oscillating the weight change will be reduced. If the weight changes in the same direction for several iterations, then the magnitude of the weight change will be increased.

Quickprop: This is also one of the better training algorithms and is loosely based on Newton's method for finding the root of a quadratic function. Standard backprop calculates the weight change based on the first derivative of the error with respect to the weight. If the second derivative is also available then better optimum step and the direction can be found. The Quickprop modification is an attempt to estimate and utilize second derivative information. Quickprop requires saving the previous gradient vector as well as previous weight change. The calculation of weight change uses only information associated with the weight being updated.

$$\Delta W_{ij} = \frac{\nabla w_{ij}(n)}{\left[\nabla w_{ij}(n-1) - \nabla w_{ij}(n)\right] \Delta w_{ij}(n-1)} \tag{3.30}$$

where,

$\nabla w_{ij}(n)$ = the gradient vector component associated with weight w_{ij} in step n.

$\nabla w_{ij}(n-1)$ = the gradient vector component associated with weight w_{ij} in previous step.

$\Delta w_{ij}(n-1)$ = weight change in (n-1)th step.

A maximum growth factor μ is used to limit the rate of increase of the step size like:

If $\Delta w_{ij}(n) > \mu \; \Delta w_{ij}(n-1)$

Then $\Delta w_{ij}(n) = \mu \; \Delta w_{ij}(n-1)$

Fahlman suggested an empirical value 1.75 for μ.

There are some complications in this method. First the step size calculation requires a previous value, which is not available at the start. This is overcome by using standard back-propagation method for the weight adjustment. The gradient descent weight change is given by

$$\Delta w_{ij}(n+1) = w_{ij}(n) - \eta \; \nabla w_{ij} \tag{3.31}$$

Value of η is taken suitably small.

Second problem is that the weight values are unbounded. They become too large that they cause overflow in the computer.

Supersab: The Self-Adjusting Back-Propagation Algorithm(SuperSAB) was developed by Tom Tollenaere. It is not as fast as quickprop or rprop with the standard problems however in this stage its too early to pass a verdict on this algorithm.

Cascade Correlation: Cascade Correlation by Fahlman and Lebiere (School of Computer Science, Pittsburgh) may be the fastest available training method and it constructs the network by adding hidden units one at a time. On of the reason it is fast is that only the new weights are trained, the rest of the weights stay fixed. It does not work very well with the function approximation problems. It begins with a minimal network and then automatically trains and adds new hidden units on by one, creating a multi-layer structure. Once a new hidden unit has been added to the network, its input-side weights are frozen. This unit then becomes a permanent feature-detector in the network, available for producing outputs or for creating other, more complex feature detectors. This architecture learns very quickly, the network determines its own size and topology, it retains the structure it has built even if the training set changes, and it requires no back-propagation of error signals through the connections of the network.

Conjugate Gradient Methods : The basic back-propagation algorithm adjusts the weights in the steepest descent direction(negative of the gradient). This is the direction in which the performance function is decreasing most rapidly. It turns out that, although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence. In the conjugate gradient algorithms a search is performed along conjugate directions, which produces faster convergence than steepest descent directions. In most of the training algorithms the learning rate is used to determine the length of the weight update (step size). In most of the conjugate gradient algorithms the step size is adjusted at each iteration. A search is made along the conjugate gradient direction to determine the step-size which will minimize the performance function along that line. There are various search functions any of these can be used interchangeably with a variety of training functions. The conjugate gradient algorithms implemented in the MATLAB toolbox are: Traincgf, Traincgp, Traincgp, Trainscg.

All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = - g_0$$

A line search is then performed to determine the optimal distance to move along the current search direction.

$\mathbf{w}_{k+1} = w_k + \alpha_k\mathbf{p}_k$, where α_k is the learning rate.

Then the next search direction is determined such that it is conjugate to previous search directions. The general procedure to determine the new search direction is to combine the new steepest descent direction with the previous search direction.

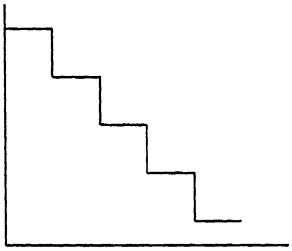
$$\mathbf{p}_k = -g_k + \beta_k\mathbf{p}_{k-1}$$

The various versions of conjugate gradient are distinguished by the manner in which the constant β_k is computed.

Variable Learning Rate Algorithms: With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm may oscillate and become unstable. If the learning rate is too small, the algorithm will take too long to converge. It is not practical to, determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface. The performance of the steepest descent algorithm can be improved if we allow the learning rate to change during the training process. An adaptive learning rate will attempt to keep the learning step size as large as possible while keeping learning stable. The learning rate is made responsive to the complexity of the local error surface. First, the initial network output and error are calculated. At each epoch new weights and biases are calculated using the current learning rate. New outputs and errors are then calculated. If the new error exceeds the old error by more than a predefined ratio, the new weights are discarded and the learning rate is decreased. If the new error is less than the old error then the learning rate is increased. This procedure increases the learning rate, but only to the extent that the network can learn without large error increases. TRAINGDA and TRAINGDX are the two variable learning rate algorithms employed in the MATLAB

TOOLBOX. Some examples of variable learning rate algorithms are discussed in the next two sub-sections.

Gradient Range Based Heuristic(GRBH) Method : It is a method which uses a few sets of learning rates instead of one fixed value, as shown if figure 3.7. The gradient values are divided up into a number of groups according to their values. For each group different learning rate is assigned. Large learning rates are assigned to groups that have small modulus value of gradient, small learning rates are assigned to those with large modulus values of gradient.



x-axis: Modulus of gradient, $\left|\frac{\partial E}{\partial w}\right|$; y-axis: Learning rate(η)

Figure 3.7: GRBH Method

The values of learning rate for each group is chosen at the beginning of the training procedure and kept constant. During the weight update procedure, the learning rate for each connection weight is found by determining to which group the gradient belongs. The gradient of each weight changes with every iteration, hence the learning rate changes with every iteration. Also, different weights have different gradients and therefore each weight will have a different value of learning rate.

Inverse Gradient Method: In this method the learning rate is calculated as a fraction of the inverse of the gradient. Learning rate is given as:

$$\eta = \frac{k}{\left|\frac{\partial E}{\partial w}\right|}, \text{ where k is a scaling coefficient} \qquad (3.32)$$

Therefore the weight change is calculated as follows:

$$\mathrm{w_{ij}(n+1) = w_{ij}(n) + \eta} \frac{\partial E}{\partial w_{ij}} + \beta \ \Delta w_{ij}(n) \tag{3.33}$$

where β is the momentum term

Application of GRBH and IG algorithms, on XOR and Alphabet recognition problems, proved that they perform better than the standard back-propagation.

Delta-Bar-Delta algorithm : The Delta-Bar-Delta is a method that implements four heuristics regarding gradient descent. It was developed by Jacobs (1988). The method consists of a weight update rule and learning update rule. The weight update rule is applied to each weight $w_{ij}(n)$ at iteration n through the relationship given by

$$w_{ij}(n+1) = w_{ij}(n) - \eta_{ij}(n+1)\frac{\partial E(n)}{\partial w_{ij}(n)} \tag{3.34}$$

where $\eta(n)$ is the learning rate for the weight $w_{ij}(n)$ at update iteration n.

The learning rate update rule for a given weight $w_{ij}(n)$ is defined as

$$\Delta\eta_{ij}(n) = \begin{cases} k & ; if\, \bar{\delta}_{ij}(n-1) \times \delta_{ij}(n) > 0 \\ -\phi\eta_{ij}(n) & ; if\, \bar{\delta}_{ij}(n-1) \times \delta_{ij}(n) < 0 \\ 0 & ; otherwise \end{cases} \tag{3.35}$$

where

$$\delta_{ij}(n) = \frac{\partial E(n)}{\partial w_{ij}(n)} \tag{3.36}$$

the partial derivative of the error with respect to $w_{ij}(n)$ at iteration n, and

$$\bar{\delta}_{ij}(n) = (1-\theta)\delta_{ij}(n) + \theta\bar{\delta}_{ij}(n-1) \tag{3.37}$$

where k and ϕ are constants used increment or decrement the learning rate respectively , and $0 < \theta < 1$ is an exponential "smoothing" base constant for the n^{th} iteration.

The heuristics implemented are as follows:

1. Every parameter (weight) has its own individual learning rate.
2. Every learning rate is allowed to vary over time to adjust to changes in the error surface.
3. When the error derivative for a weight has the same sign for several consecutive update steps, the learning rate for that weight should be increased. This is because the error surface has a small curvature at such points and will continue to slope at the same rate for some distance. Therefore, the step-size should be increased to speed up the downhill movement.
4. When the sign of the derivative of a weight alternates for several consecutive steps, the learning rate for that parameter should be decreased. This is because the error surface has a high curvature at that point and the slope may quickly change sign. Thus, to prevent oscillation, the value of the step-size should be adjusted downward.

3.4 Improving Results

Faster training is great but ultimately you want the best possible results on a test set. This section discusses the techniques involved in achieving better results. The possible options are :

- Over-fitting
- Bad Generalization
- The Size of the Network
- Combining Network Outputs

- Weight Decay
- Acceleration Algorithms
- Weight Pruning
- Pruning Hidden Layer Units
- Extracting Rules

3.4.1 **Over-fitting:** Backprop can fit any function even if the form of the function, to be fitted, is not known. This leads to over-fitting, as the training goes on the network will end up fitting the training set data very closely while running the results on the test set. Figure 3.8 indicates the problem of tying to fit the line y = x + 1 (Donald R Tveter, Pattern Recognition Basis of Artificial Intelligence)). The points marked with asterisk are training set points set just above and below the exact line, this is typical of real world measurements, the deviation from ideal is said to be result of noise in the data. The best fit came at 2700 iterations and then the over-fitting began. Quite often people who want to fit a function want to extrapolate beyond the range of the training data and this could be disastrous. As seen in the figure beyond x =1 the slope has gone down to nearly 0 instead of being close to 1. Even at 2700 iterations the network has placed its line very close to the left and right data points thus even results near the edge of the range of the training data are often poor.

3.4.2 **Bad Generalization:** In a pattern classification problem there is no guarantee that the backprop network is going to come up with a sensible way to partition the boundaries between the classes. The following is an example considered by D Tveter . Figure 3.9(a) shows two linearly separable classes with four points each marked A and B, a straight line separates them. Figure 3.9(b) shows linearly non-separable classes(created by adding two points in each class), it shows the curve required to separate them. Figure 3.9 show the boundaries created when backprop was used. In most of the cases the training resulted in networks like figure3.9(c). Sometimes it came up with stranger solutions like 3.9(d) and 3.9(e) . As can be seen from the lower portion of figure 3.9(e), it has been classified as class A whereas no points belonging to A is present there. Such odd generalization can be minimized by averaging results over a number of networks.

3.4.3 **The Size of the Network:** It is widely stated that you will get best results on the test set wit a relatively small number of hidden neurons. In fact one rule is that you need at least as many training set pattern as there are weights in the network. On the other hand there are reports that in certain cases networks with many more weights than patterns also work well.

3.4.4 **Combining Network Outputs:** One of the techniques used to improve results is to combine estimators (network outputs) of many networks, the equivalent of getting the opinion of many experts. Suppose the solution to a problem is a straight line. A network is unlikely to find that straight line, it will find a curvy line near the straight line. Probably another network will find a different curvy line near the line. By averaging a number of such results the curve may tend to cancel out and it may give a straight line. Further there have been experiments that show that giving the results from one network a higher weight than another network also helps. One advantage of these methods is that they give good results even with rather poorly trained networks.

3.4.5 **Weight Decay:** If the form of the function being fitted is known then it is better if some standard statistic routine is used on it. For example if the function is linear one can do a simple least square analysis and find the answer faster than by using backprop. With backprop the network does not know the form of the function and it may overfit the function. This can be avoided by using smallest possible number of hidden layer units. Another way is to use weight decay, a method that tries to keep the weight small. The idea is to subtract a small fraction from each weight at every pass through the network. If the weight is w and the tuiny fraction is lambda * w, then use w = w – lambda * w .

3.4.6 **Weight Pruning:** Another method to improve results is to prune away excess weights. The general architecture of Neural Network model contains a large number of artificial neurons and weight connections for carrying various operations. The exact number of hidden nodes or the connections required is very difficult to formulate.

Researchers have indicated that less number of neurons result in inappropriate modeling of the input-output. On the other hand, a large number of neurons result in increased hardware implementation cost and complexity in manufacturing. Similarly, the multi-input systems require a large number of inputs. Some of these have less contribution towards the input-output behavior of the system. Weight pruning method suggests deleting the weights, which have less importance in the network starting from a large sized network. The three algorithms available for this purpose are:

- Penalty function methods
- Sensitivity analysis method
- Iterative pruning method

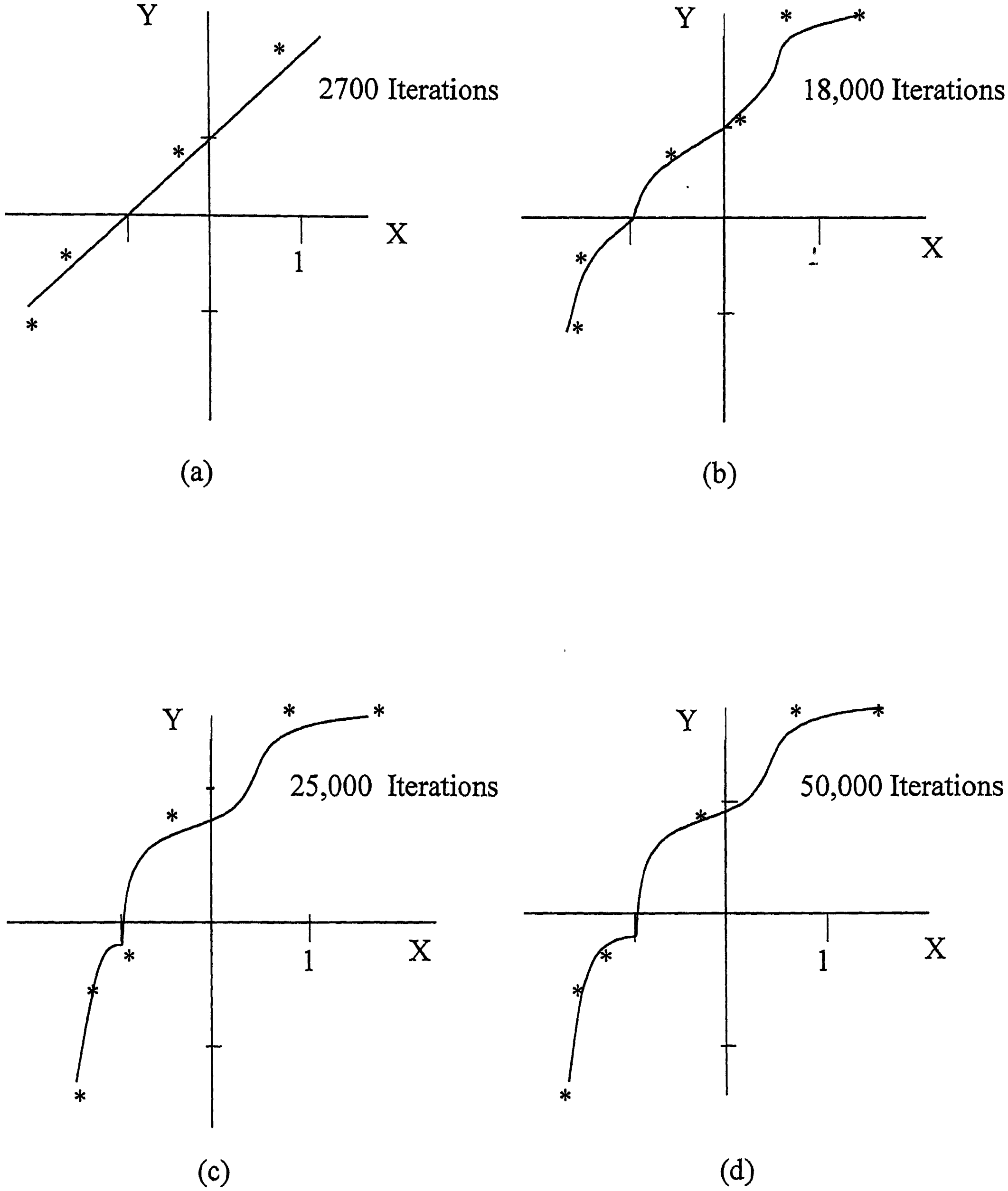
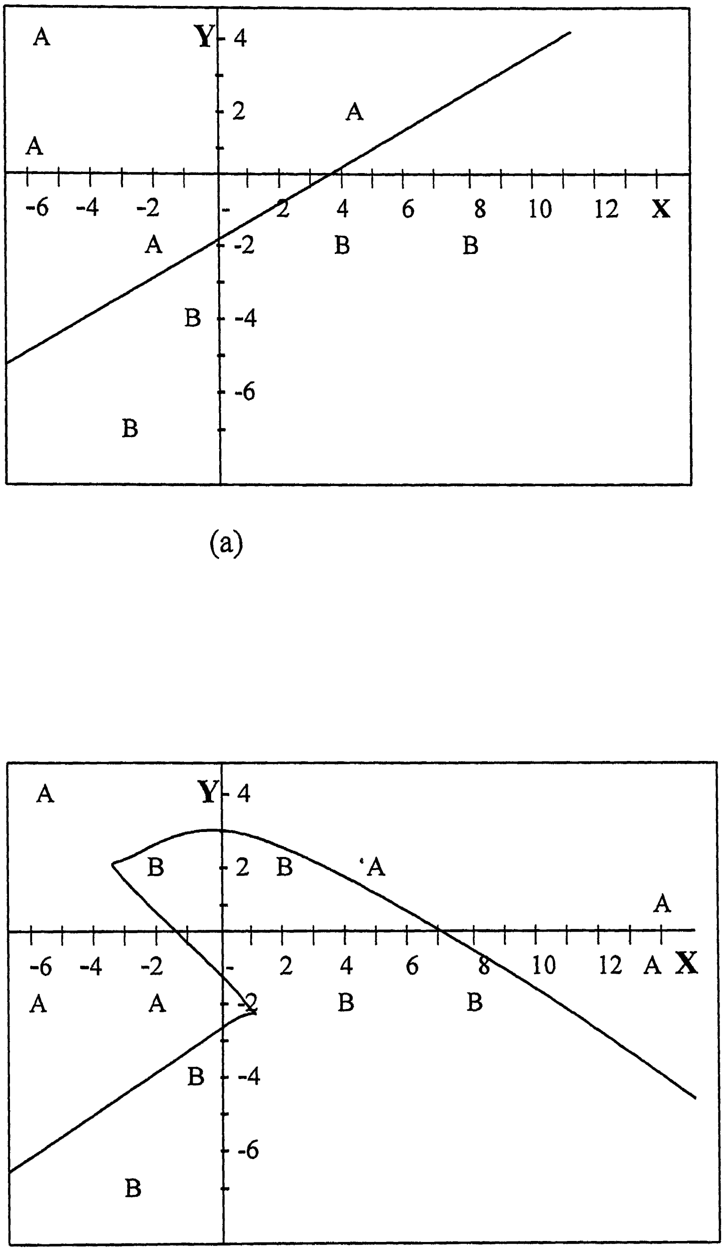


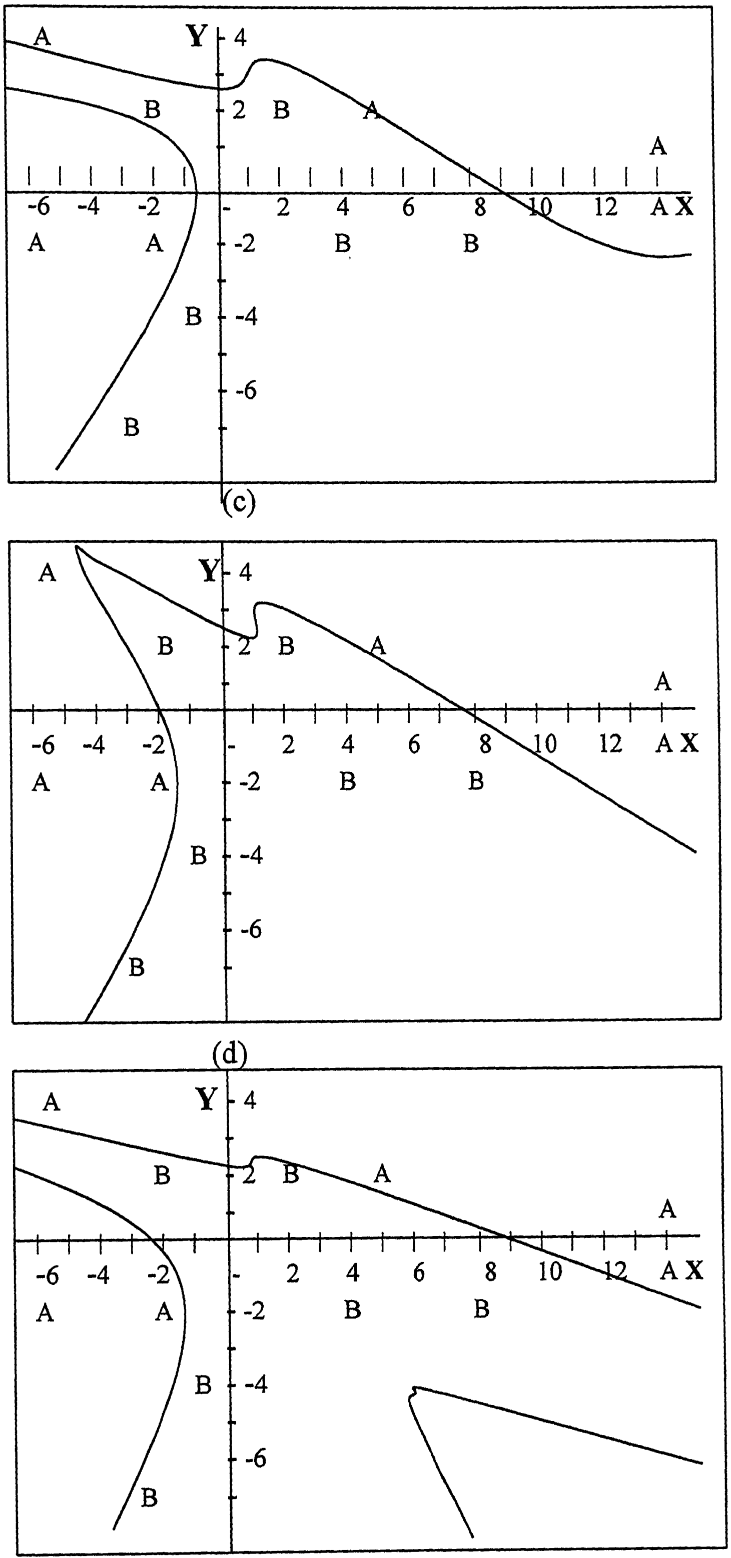
Figure 3.8: Plots showing Over-fitting (for y = x + 1)



(a)

(b)

Figure 3.9 : Bad Generalization



(c)

(d)

(e)

Figure 3.9 (contd): Bad Generalization

Chapter 4

Learning

One of the most significant attributes of neural network is its ability to learn from its surrounding, and thus improve its performance. Learning takes place by an adaptive process, known as *learning rule* or *algorithm,* whereby the weights of the network are adjusted so as to improve the performance. Learning process can be viewed as an optimization process, or a search in the weight space for a solution. The learning rules can be classified into supervised, unsupervised or reinforcement learning. Various learning algorithms from each of these categories have been implemented in the design of neural network. A prescribed set of well defined rules for the solution of a learning problem is called a *Learning Algorithm* . Basically the learning rules differ from each other in the manner in which they adjust the weights of the synapses.

4.1 Various Learning Paradigms: The three learning paradigms are as follows:

i) Supervised learning

ii) Reinforcement learning

iii) Unsupervised learning

4.1.1 **Supervised Learning**: In this form of learning each input pattern is related to a specific desired output, as shown in figure 4.1. The weights are adjusted at each step to reduce the difference between the desired and the actual output. This learning can be performed either on-line or off-line. Back-propagation algorithm uses supervised learning. The basic disadvantage of this method is that without a teacher it cannot learn new strategies.

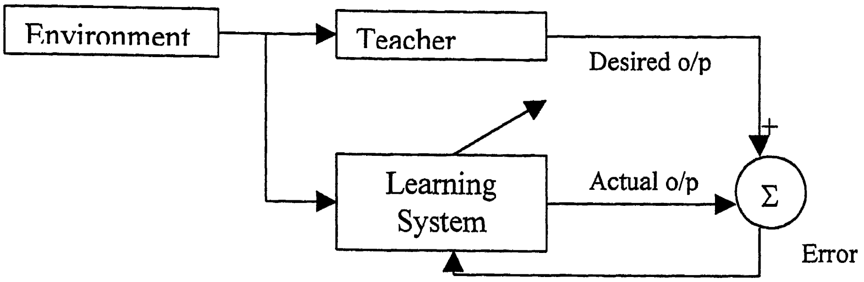


Figure 4.1 : Supervised Learning

4.1.2 **Unsupervised Learning :** This is also referred to as self-organized learning. This requires no teacher or critic to oversee the learning process. There are no specific examples of the function to be learned by the network. The aim here is to optimize some performance function defined in terms of output activity of the units in the network. Once the network is tuned to the statistical regularities of the input data, it develops the ability to form internal representations for encoding features of the input and thereby create new classes automatically. Competitive learning is a type of unsupervised learning.

4.1.3 **Reinforcement Learning** :It involves updating the weights in response to an "evaluative" teacher signal; it is different from supervised learning where the teacher signal is the "correct answer." Its on-line learning of an input-output mapping through a process of trial and error designed to maximize a *scalar performance index* called a reinforcement signal. The basic idea behind this learning is to learn an evaluation function, so as to predict the cumulative discounted reinforcement to be received in the future.

4.2 Various Learning Algorithms

Neuron is an adaptive element, its weights are modifiable depending upon the input signal it receivers, its output value and the associated teacher response, as shown in figure 4.2. In some cases the teacher signal is not available and no error information can be used thus neurons will modify wt, based on input and/or output.

$\underline{W}_i$ = weight vector ; w_{ij} = weight between j^{th} input and i^{th} neuron

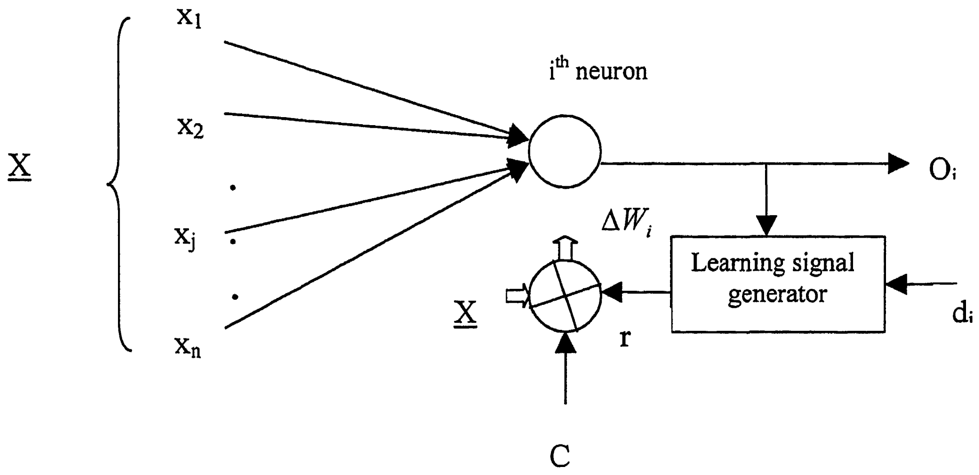


Figure 4.2 : Supervised Learning

Under different, learning rules, the form of the neuron's *Activation Function* may be different. Threshold parameter may be included in learning as one of the weight. This would require fixing one of the inputs, say x_n. We assume here that x_n is fixed and x_n =-1

A general learning rule adopted in neural network

$\underline{W}_i = [w_{i1}, w_{i2} ....w_{in}]^t$ increases in proportion to the product of input $\underline{X}$ and learning signal r. r is a function of $\underline{W}_i$ and $\underline{X}$ and sometimes of teacher's signal d_i

Therefore,

$$r = r\,(\underline{w}_{i1}, \underline{x}, di) \qquad (4.1)$$

$$\Delta\ \underline{W}_i(t) = Cr\ \{\underline{W}_i(t), \underline{X}(t), di(t)]\ \underline{X}(t) \qquad (4.2)$$

When C is a positive number called *learning constant*, it determines the rate of learning.

Weight at t+1 will be thus

$$\underline{w}_i(t+1) = \underline{w}_i(t) + Cr\left\lfloor \underline{w}_i(t), \underline{x(t),}d_i(t)\right\rfloor \underline{x}(t) \qquad (4.3a)$$

using discrete time training steps

$$\underline{w}_i^{k+1} = \underline{w}_i^k + cr(\underline{w}_i^k, \underline{x}^k, di^k)\, \underline{x}^k \tag{4.3b}$$

The learning so far has been a sequence of discrete-time weight modification continuous-time learning can be expressed as

$$\frac{d\underline{W(t)}_i}{dt} = Cr\underline{X}(t) \tag{4.4}$$

Her we assume that the weights have been suitably initialized before each learning experiment was started. The various learning algorithms implemented in the design of neural network can be listed as follows:

i) Error-correction learning

ii) Delta learning rule

iii) Hebbian learning

iv) Competitive learning

v) Boltzmann learning

vi) Widrow-Hoff learning

4.2.1 **Error Correction Learning** : This rule was originally proposed for a single-unit training. This rule drives the output error to zero. Let

d_k (n) = Desired response of neuron k at time n

y_k(n) = Actual response

$\underline{X}$(n) = Input to the Network

The error signal is given by:

$$e_k(n) = d_k(n) - y_k(n) \tag{4.5}$$

The ultimate aim of error correction learning is to minimize a *cost function* based on the error signal e_k(n) such that the actual response of each output neuron approaches the target response. The criterion used for selection of *cost function* is *mean square error criterion,* defined as the mean-square value of the *sum of squared errors :*

$$J = E\left[\frac{1}{2}\sum_{k} e_k^2(n)\right] \tag{4.6}$$

Where, E = Statistical expectation operator

$\sum_{k}$ = Summation over output neurons

J has to be minimized with respect to the free parameters. Minimization of cost function(J) with respect to the network parameters leads to the so called *method of gradient descent.* Another easier method or cost function or criteria uses instantaneous value of the sum of square error:

$$\varepsilon(n) = \frac{1}{2}\sum_{K} e_k^2(n) \tag{4.7}$$

The network is optimized by minimization of $\varepsilon(n)$ with respect to the synaptic weights, as per the error correcting learning rule (or delta rule) the adjustments, $\Delta w_{kj}(n)$ made to $w_{kj}(n)$ is given by:

$$\Delta w_{kj}(n) = \eta\ e_k(n)\ x_j(n) \tag{4.8}$$

Where $\eta$is the learning rate, it's a positive constant. Thus from the above equation we see that the adjustments in the synaptic weights is proportional to the product of error signal and the input signal of the synaptic in question.

Thus,

$$e_k(n) = d_k(n) - y_k(n)$$

$$\Delta w_{kj}(n) = \eta\ e_k(n)\ x_j(n)$$

$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n) \tag{4.9}$$

Error correction learning is a closed feedback system. Here the choice of η is of great significance, a very small value would give a smooth convergence, but it will consume more time, whereas a high value of η would give quick convergence but it involves the danger of divergence.

4.2.2 **Delta Learning Rule** : (Supervised) : This is valid only for continuous activation functions, and it is supervised learning scheme, as shown in figure 4.3. The learning signal for this rule is called *delta* and is defined as:

$$r \underline{\underline{\Delta}} \left[d_i - f(\underline{w}_i^t \underline{x})\right] f'(\underline{w}_i^t \underline{x}) \qquad (4.10)$$

$f'(\underline{w}_i^t \underline{x})$ = derivative of activation function f(net), where $net = \underline{w}_i^t \underline{x}$

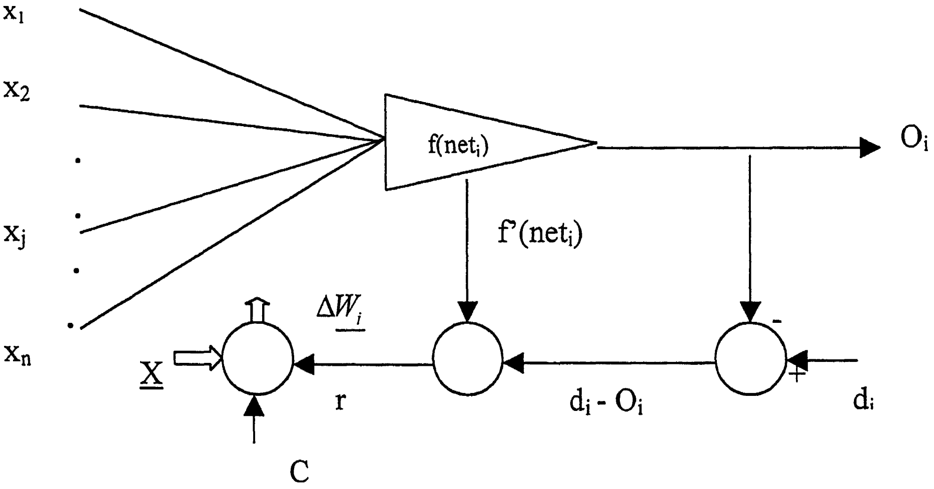


Figure 4.3 : Delta Learning

This learning rule is derived from condition of *least square error* between Oi and d_i. Calculating the gradient vector w.r.t. $\underline{w}_i$ of the squared error defined as:

$$E = \frac{1}{2}(d_i - o_i)^2 \qquad (4.11a)$$

It is equivalent to

$$E = \frac{1}{2}[d_i - f_i(\underline{w}_i^t \underline{x})]^2 \qquad (4.11b)$$

the *error gradient vector* value is given by

$$\nabla E = -(d_i - o_i) f'(\underline{w}_i^t \underline{x}) \underline{x} \qquad (4.12a)$$

the components of the gradient vector are

$$\frac{\partial E}{\partial w_{is}} = -(d_i - o_i)\, f'(\underline{w}_i^t \underline{x})\, x_j \quad \text{for j = 1, 2 ....n} \qquad (4.12b)$$

Minimization of error requires the weight changes to be in the negative gradient direction, thus we take

$$\Delta \underline{w}_i = -\eta \nabla E \qquad (4.13)$$

η = +ve constant

from equation 4.12a and 4.13 we have

$$\Delta \underline{w}_i = \eta (d_i - o_i)\, f'(net_i)\, \underline{x} \qquad (4.14a)$$

and for a single weight adjustment

$$\Delta w_{ij} = \eta (d_i - o_i)\, f'(net_i)\, x_j \quad \text{for j = 1, 2........n} \qquad (4.14b)$$

Therefore, the weight adjustment in the above equation is computed based on minimization of the squared error. Using the general learning rule, we have

$$\Delta \underline{w}_i(t) = Cr\left[\underline{w}_i(t),\ \underline{x}(t),\ d_i(t)\right]\underline{x}(t)$$

and plugging the learning signal given by 2.36., the weight adjustment becomes

$$\Delta \underline{w}_i = c(d_i - o_i)\, f'(net_i)\, \underline{x} \qquad (4.15)$$

which is similar to (4.14), since c and η are arbitrary constants. Thus,

⟶ The weights are initialized at any values for this method of training

⟶ This rule is also called the Continuous Perceptron Training Rule

⟶ Delta Learning Rule can be generalized for multi-layer networks

Delta learning rules requires the calculation of $f'(net)$ at every step. Therefore, we use these equations:

$$f(net) = \frac{2}{1+\exp(-net)} - 1;$$ Continuous bipolar activation function

$$f'(net) = \frac{2\exp(-net)}{[1+\exp(-net)]^2}$$

$$O = f(net)$$

Therefore,

$$\frac{1}{2}(1-O^2) = \frac{1}{2}\left[1-\left(\frac{1-\exp(-net)}{1+\exp(-net)}\right)^2\right]$$

$$= \frac{2\exp(-net)}{[1+\exp(-net)]^2} = f'(net)$$

Therefore, $f'(net) = \frac{1}{2}\left(1-O^2\right)$ (4.16)

Equation 4.16 is valid for bipolar continuous activation functions.

4.2.3 **Hebbian Learning :** "If two neurons on either side of a synapse are activated simultaneously then the strength of that synapse is selectively increased. If the two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated". Such a synapse is called *Hebbian Synapse* . Refers to figure 4.4.

According to Hebbian learning, the adjustments applied to w_{kj} at time n is expressed in the form:

$$\Delta w_{kj}(n) = F(y_k(n), x_j(n)) \qquad (4.17a)$$

F(. , .) is a function of both post-synaptic and pre-synaptic activities. As a special case we can write :

$$\Delta w_{kj}(n) = \eta\, y_k(n)\, x_j(n) \qquad (4.17b)$$

This rule shows the correlation nature of the Hebbian synapse $w_{kj,}$ expressed as product of incoming and outgoing signals. There are modifications to the synaptic adjustment to ensure that it does not saturate w_{kj}. The modifications are as follows:

$$\Delta w_{kj}(n) = \eta\, y_k(n)\, x_j(n) - \alpha\, y_k(n)\, w_{kj}(n) \qquad (4.18)$$

$$\Delta w_{kj}(n) = \alpha\, y_k(n) \lfloor c x_j(n) - w_{kj}(n) \rfloor \qquad (4.19)$$

$$c = \frac{\eta}{\alpha}$$

This equation implies that for inputs for which $x_j(n) < w_{kj}(n) / c$, $w_{kj}(n+1)$ will reduce(at time n+1) by an amount proportional to postsynaptic activity $y_k(n)$ and vice versa as shown in the graph.

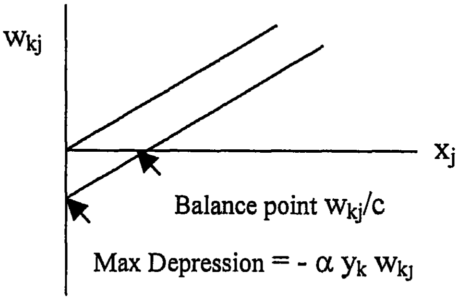


Figure 4.4: Hebbian Learning Graph

The activity balance point for modifying the synaptic weight at time n+1 is a variable, w_{kj}/c, it is proportional to the value of w_{kj} at the time of pre-synaptic activity. This approach eliminates the problem of runaway synaptic weight instability and results in negatively accelerating modification curve. Another way of formulating the Hebbian postulate is to make the change in the synaptic weight proportional to covariance between pre-synaptic and post-synaptic activities. According to this rule, on an average, the strength of the synapse increases if the post-synapse and the pre-synapse activities are positively correlated and decreases if negatively correlated.

4.2.4 **Competitive Learning:** The output neurons of a network compete among themselves for being the one to be activated or fired. At a time only one output neuron can be active, as shown in figure 4.5. The three basic elements to a competitive elements are:

- A set of neurons that are all same except for randomly distributed weights, and therefore respond in different manner to a given set of input patterns.
- A "limit" imposed on the "strength" of each neuron.
- A mechanism that permits the neurons to compete for the right to respond to a given input/s, such that only one output neuron is active at a time and this neuron is called " Winner-takes-all" neuron.

Thus the individual neurons of the network learn to specialize on sets of similar patterns, and thereby become *feature detectors.*

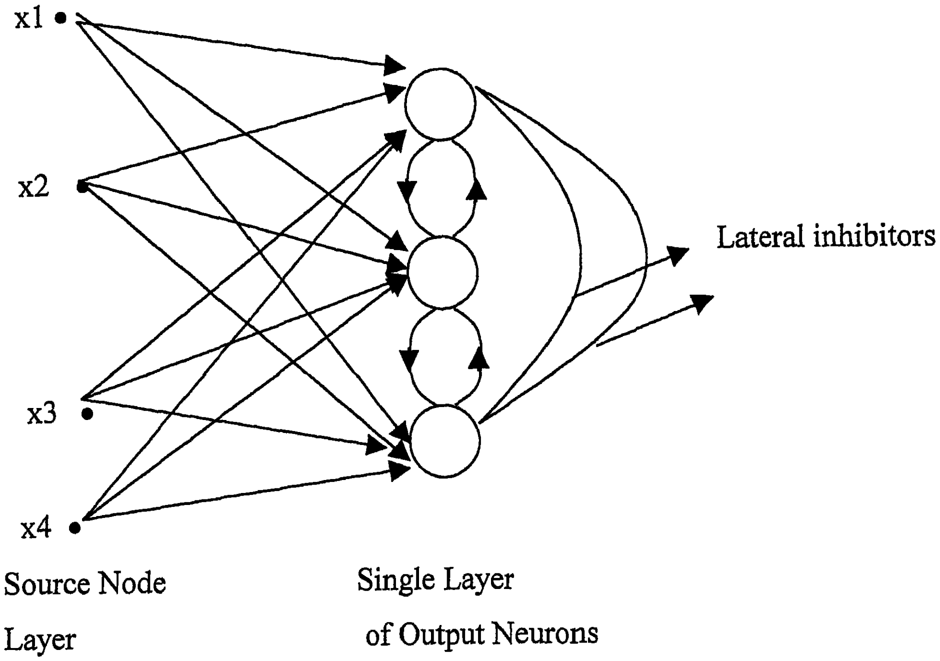


Figure 4.5: Competitive Learning Network

If the internal activity level for a given input, say x, is the largest, then it is the winning neuron. Its output is made 1 and output of the rest are 0.

$$\sum_{i} w_{ji} = 1 \quad \text{for all j (output neurons)} \qquad (4.20)$$

$$\Delta w_{ji} = \begin{cases} \eta(x_i - w_{ji}) & \text{if neuron j wins the competition} \\ 0 & \text{if neuron j loses the competition} \end{cases} \qquad (4.21)$$

4.2.5 **Boltzman Learning :** It's a stochastic learning algorithm. The neurons constitute a recurrent structure and they operate in binary manner, i.e. they are either "on"(+1) or "off"(-1). The machine is characterized by energy function <u>E.</u>

$$E = -\frac{1}{2}\sum_{i}\sum_{j} w_{ji}\, s_i\, s_j \qquad (4.22)$$

Where, s_i = state of neuron I

w_{ji} = weight of link between neuron I and j

$i \neq j$; no feedback

4.2.6 The Widrow-Hoff learning rule

The Widrow-Hoff learning rule (Widrow, 1962) is applicable for supervised learning of neural networks. It is independent of the activation function of the neurons since it minimises the squared error between the desired output value d_j and the neuron's activation value $net_j = \mathbf{w}_j^t\mathbf{x}$. The learning signal for this rule is defined as follows:

$$r \,\underline{\underline{\Delta}}\, d_j - net_j \qquad (4.23)$$

The weight vector increment under this learning rule is

$$\Delta\overline{w}_j = c\left(d_j - net_j\right)\overline{x} \qquad (4.24)$$

or for the single weight adjustment is

$$\Delta w_{ij} = c\left(d_j - net_j\right)x_i \qquad ; for\ i = 1,2,\ldots\ldots\ldots n \tag{4.25}$$

This rule can be considered a special case of the delta learning rule with linear activation function i.e. in the equation (4.10), *f(net)* = *net* and *f'(net)* = 1 and the subject equation becomes identical to equation (4.23). This rule is, sometimes called the *LMS (least mean square) learning rule*. Weights are initialised at any values in this method.

Chapter 5

Benchmark Problems

There are various benchmark problems that have been developed to analyse the performance of multi-layered feed-forward neural networks. These problems are different from the common problems encountered, in the sense that they cover various aspects that are required to be tested for before branding any algorithm as successful. Researchers to test various performance characteristics of Multi-layer Feed-forward networks and to verify theoretical results through simulation use these typical problems. In this thesis, an attempt has been made to solve these benchmark problems as well as a few more typical model problems in classification and function approximation areas using multi-layered feed-forward networks. These problems have been solved with network models developed using MATLAB neural network toolbox. Further, these solutions have been compared with those obtained using models developed in JAVA language. The codes written in JAVA need continuous refinement to match the performance of MATLAB.

5.1 Exclusive-OR (XOR) Problem

A single layer perception has no hidden layer. It cannot classify the input patterns that are not linearly separable. XOR is an example of linearly non-separable, as shown in figure 5.1. It can be viewed as a special case of a more general problem, namely, that of classifying point in a unit hypercube (XOR is the case of dimension).

Each point in the hypercube belongs to class 0 or class 1.

Input Pattern		Output pattern
0	0	0
0	1	1
1	0	1
1	1	0

$$\left.\begin{array}{l}0 \text{ XOR } 1 = 1\\ 1 \text{ XOR } 0 = 1\end{array}\right\}\begin{array}{l}\text{i.e. (1,0) and (0,1)}\\ \text{belong to class 1}\end{array}$$

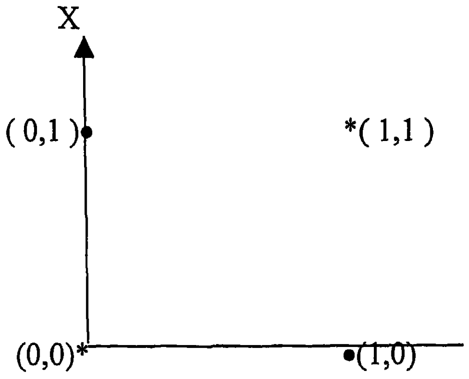


Figure 5.1: Exclusive-OR(XOR)

The use of a single neuron with two inputs results in a straight line for the decision boundary in the input space. For all the points on one side of this line, its output is 1 and for the points on the other side, the neuron output is 0. The position and the orientation of the line depend on the weight and the threshold. In case of XOR the decision boundary cannot be a straight line. Elementary perceptron cannot solve the XOR problems.

XOR can be solved by using a hidden layer with two neuron, as shown in figure 5.2.

--Each neuron is represented by McCulloh – Pitts model.

--Bits 0 and 1 are represented by 0 & +1.

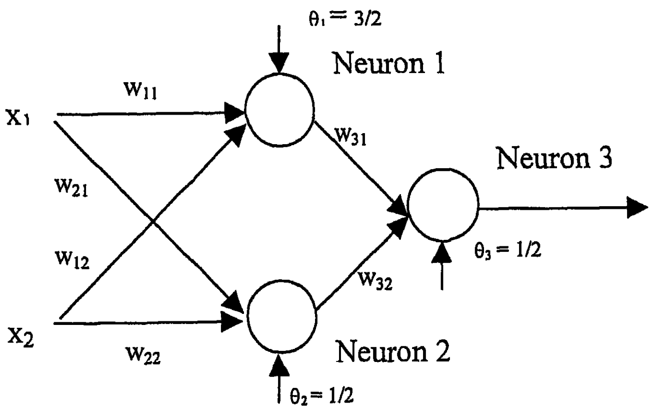


Figure 5.2: Network to solve XOR

The top neuron (neuron 1) is characterized as follows:

$W_{11} = W_{12} = + 1$; $\theta_1 = + 3/2$

The bottom neuron (neuron2) is characterized as follows:

$W_{21} = W_{22} = 1$; $\theta_2 = +1/2$

The output neuron (neuron 3) is characterized as follows:

$W_{31} = -2$

$W_{32} = +1$

$\theta_3 = +1/2$

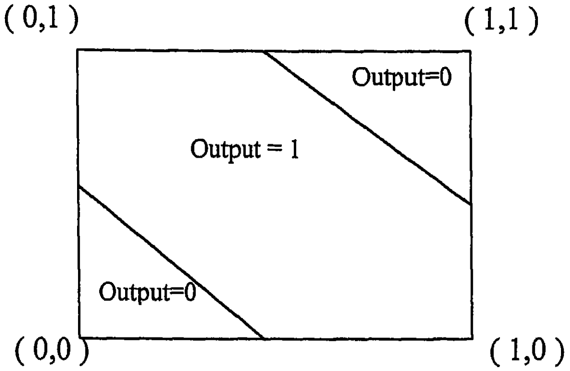


Figure 5.3 : Exclusive-OR (XOR) Decision Boundaries

Neuron 2 has excitatory (+ve) connection to the output neuron whereas Neuron 1 has inhibitory (-ve) connection to the output neuron.

- When both hidden neurons are OFF(i.e. Input = (0,0)) then the output neuron remains OFF.
- When both hidden neurons are ON (i.e. Input = (1,1)) then the output neurons is switched OFF again as the inhibitory effect of the large negative weight connected to the top hidden neuron overpowers the excitatory effect of the positive weight connected to the bottom hidden neuron.
- When top neuron is OFF (Input = (0,1) & (1,0)) and bottom neuron is ON the output neuron is switched ON due to excitatory effect of the positive weight connected to the bottom hidden neuron – Thus XOR is solved.

5.2 **The N – Parity Problem**

Problem: Mapping of N-bit wide binary number into its parity. If the input pattern contains an even number of 1^s then its parity is 1 else its 0, as shown in table 5.1. This is a difficult problem because the pattern that are closer (using Euclidean distance) in the

5.3 The Two Spiral Problem: (Proposed: Alexis Wieland of MITRE lrp.)

It's a benchmarking problem, since it is extremely hard to solve using Back propagation algorithm. It's a problem of classification. The problem consists of Np=194 (X,Y) pairs of points that lie in interlocking spirals (Np/2 on each) that goes around a common origin three times. Each spiral belongs to a different class and the network has to associate points in the sample space with the correct class (+ 1, -1). Refers to figure 5.5.

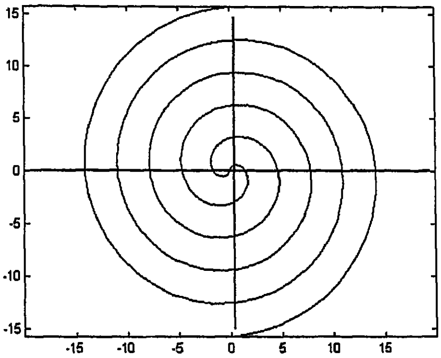


Figure 5.5: TwoSpiral

5.4 The Encoding /Decoding Problem : Ackely, Hinton and Sejnowski

posed a problem for internal representation testing in which a set of N_I orthogonal input pattern are mapped to a set of N_O orthogonal input patterns through a small set of hidden neurons ,N_H. This benchmark forces the hidden layer nodes to be efficient. The expectation is that the hidden units will form a binary encoding of the N_I- bit input pattern on to a N_H-bit pattern and then from a binary decoding of N_H-bit pattern on to a N_O-bit output pattern.

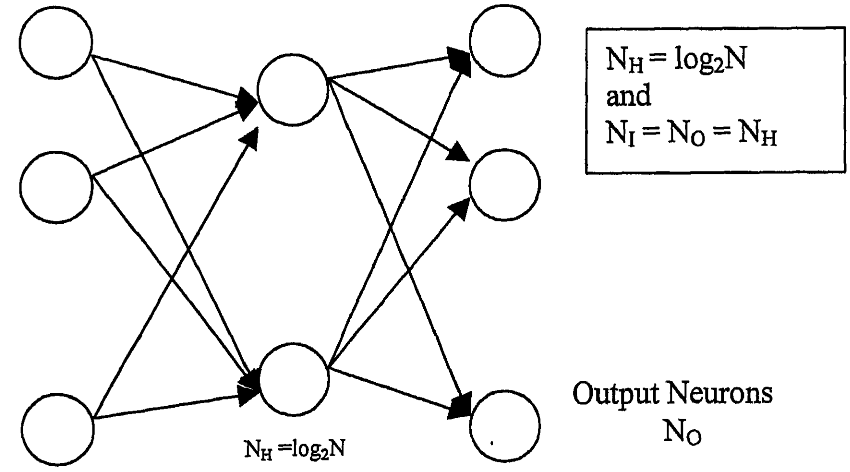


Figure 5.6: Encoding-Decoding Network

This benchmark problem is important to evaluate the network storage capacity and it has application to data compression and transmission.

5.5 Logic/Arithmetic Problem:

The logic/ arithmetic operation problems have been studied since the conception of ANN. Minsky and Papert (1969) used the XOR problem to show the limitation of single layer preceptrons. The basic symbol for combinatorial logic gates can be shown as below. By combining these gates, any logic statement can be implemented.

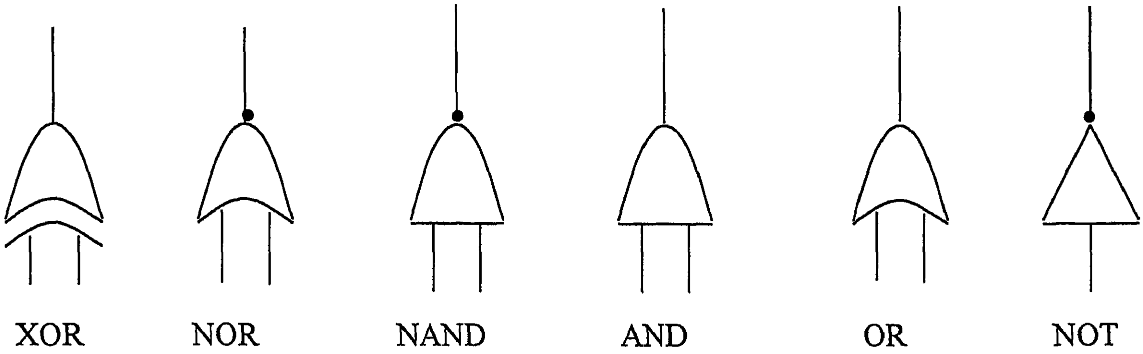


Figure 5.7: logic Gates

They can also be combined to generate temporal components useful for sequencing. ANN can also be formed that imitate the same logic elements.

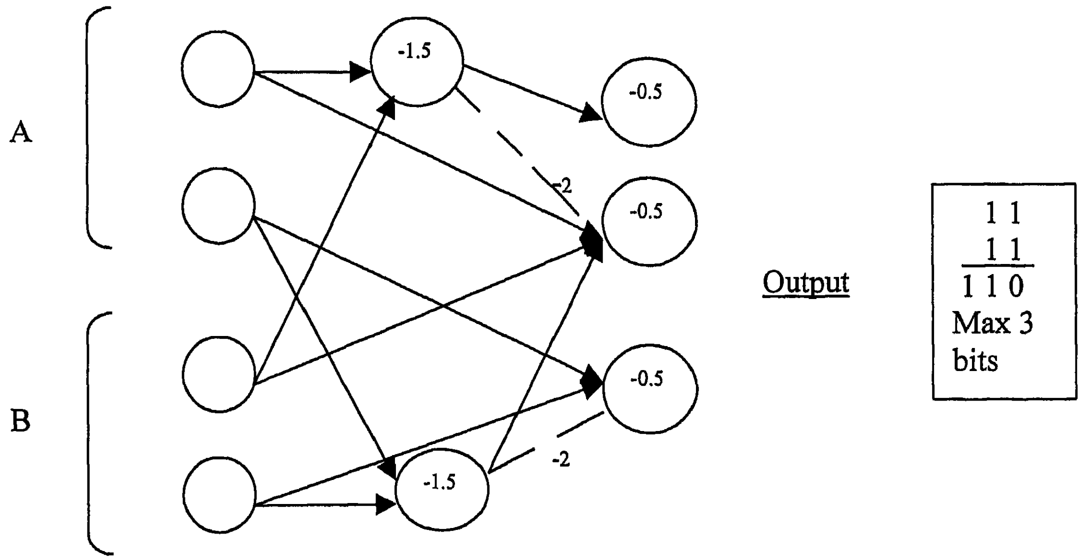


Figure 5.8:ANN minimum size adder of 2-bit numbers.

As shown in the figure 5.8, two more logic/arithmetic problems of importance are *the binary addition* and *the binary negation.* The binary addition has the feature of being able to find the local minimum. Hence it can it can be studied for network and learning algorithms to check for the manner in which they find and avoid local minima. The binary addition of a two-bit binary number gives at most a 3-bit number.

The NAND and the NOR, functions can be described by the network shown below with McCulloh Pitts models.

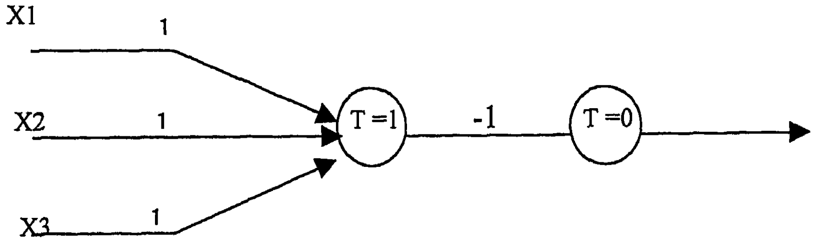


Figure 5.9: NOR

$$0^{k+1} = 1 \quad \text{If } \sum_{i=1}^{n} w_i x_i >= T$$

$$= 0 \quad \text{If } \sum_{i=1}^{n} w_i x_i < T$$

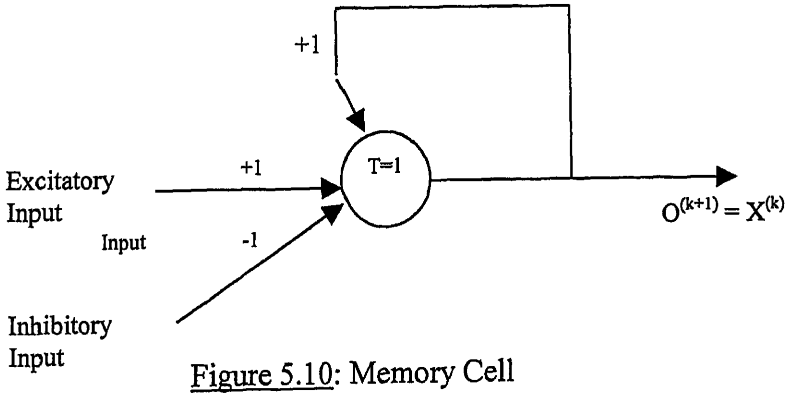


Figure 5.10: Memory Cell

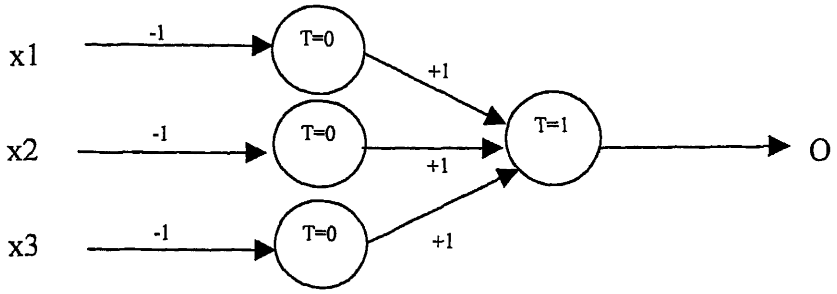


Figure 5.11: NAND Gate

The binary negation problems consists of an (n + 1) bit number $(b_0,b_0,\ldots\ldots\ldots b_n, b_{n+1})_i$ as input (n-bit data and an extra negation bit $b_N = b_{N+1}$) The output is an n-bit number $(b_0,b_1,\ldots\ldots\ldots b_n)_o$, that is equal to the input pattern if b_N is 0, and the output is the complement of input.

If bn = 1 i.e.

$(b_0, b_1,\ldots\ldots\ldots\ldots b_n)_0 = (b_0, b_1,\ldots\ldots\ldots\ldots b_n)_i$ if $b_N = 0$

$= (\overline{b_0}, \overline{b_1},\ldots\ldots\ldots\ldots \overline{b_n})_i$ if $b_N = 1$

The problem reduces to a set of n XOR problems between the negation bit and each of the inputs $(b_k)_i$ such that when $b_N = 0$

$(b_k)_o = (b_k)_i$ with $(b_k)_i = 0 \oplus b_N = 0$

$(b_k)_i = 1 \oplus b_N = 1$

And when, $b_N = 1$

$(b_k)_o = (b_k)_i$ with $(b_k)_i = 0 \oplus b_N = 1$

$(b_k)_i = 1 \oplus b_N = 0$

5.6 The Accuracy/Classification Problem:

Another measure of performance is the network accuracy. A way of testing this is to define a classification problem where, there is boundary layer between classes where the error is acceptable i.e. not penalized or less penalized.

A general geometry can be specified for the classification. The basic data is shown in the figure below where the decision boundary is fuzzy i.e. a 'Boundary layer' (BL) is defined where no training data is present and perfect recall is expected. The BL thickness, Δ c (Euclidean distance between classes) is the relative accuracy for separation for a given network. The smaller the Δ c for a network, the more accurate it can be.

This problem can be used to determine the effectiveness of the learning algorithms to converge to the desired accuracy. Conventional forms of back propagation yields a learning time that diverges like $1/\varepsilon^2$, where $\varepsilon = \frac{\Delta C}{C_{\max}}$

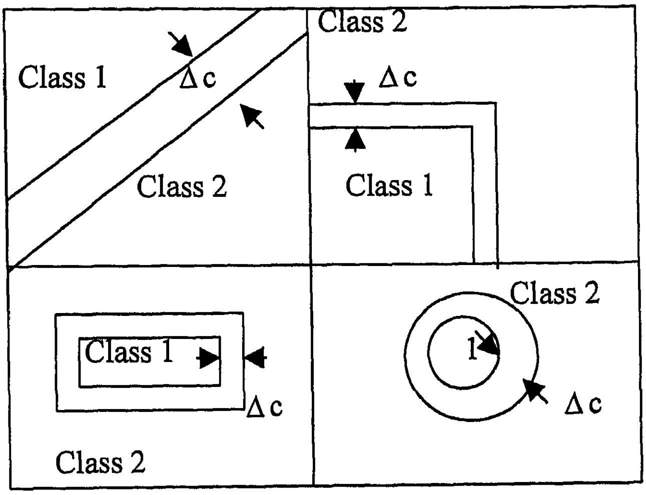


Figure 5.12: Accuracy Problem

Typically, the patterns belonging to a class are not all same, i.e. the patterns used for testing and those for training may not be the same. Pattern classification problems are said to belong to the category of supervised learning.

5.7 Majority Vote : Majority vote networks are trained to output a one when more than half of the binary inputs are 'on '(one). Otherwise, the network outputs a zero. These networks typically have an odd number *n* of inputs and a single output. The number of hidden units will vary, but typically will be less than the number of inputs.

5.8 The Sin(x)Sin(y) Problem : General function mapping problems have been used by different researches to test their network capabilities, learning algorithms etc. A popular function approximation or function mapping is:

Z(x,y) = Sin(x) Sin(y)

This function gets more complicated as the norm of the input vector(x,y) grows. It falls in the category of two-input-one-output problem. Refers to figure 5.13.

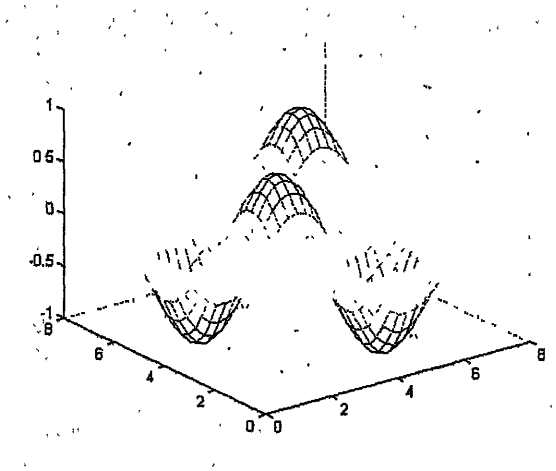


Figure 5.13: Sin(x) Sin(y) Plot

5.9 Function Approximation : A multi-layer perceptron trained with back-propagation algorithm can be viewed as a practical vehicle for performing non-linear input-output mapping of a general nature. Suppose p denotes the number of input nodes of a multiplayer perceptron, and q denotes the number of neurons in the output layer of the network. The input-output relationship of the network defines a mapping from a p-dimensional Euclidean input space to a q-dimensional Euclidean output space, which is infinitely continuously differentiable. The fundamental result states that a two layered feed-forward network with a sufficient number of hidden units, of the sigmoidal activation type, and a single linear output unit is capable of approximating any continuous function f: $R^n \rightarrow R$ to any desired accuracy. The above statement has been proved and stated in the form of various theorems(Kolmogrov's Theorem, Cybenko's theorem Hornik et al). All these people independently proved the theorem that one-hidden-layer feed-forward neural network is capable of approximating uniformly any continuous multivariate function to any desired degree of accuracy. Hornik et. al proved another important result relating to the approximating capability of multi-layer feed-forward neural networks employing sigmoidal activation functions. They showed that these networks can approximate not only an unknown function but also its derivative. These results have further been extended to any continuous function f on R^n , that these can be accurately approximated by adjusting the weights and thresholds only.

5.10 Character Recognition: Multi-layer feed-forward networks can be trained to solve a variety of data classification and recognition problems. The network can be used for handwritten as well as printed letter or alphabet or integer or symbol recognition. The input character is first normalized to extend it to full height and width of the bar mask. The characters are required to be encoded and presented to the network for training. The method of encoding the characters is of great significance, the endeavor must be to encode in a manner so as to create maximum separation between the characters. This ensures that a properly trained network can even recognize noisy data easily. In the present work recognition of integers from 0 to 9 and recognition of English letters have been considered. The encoding has been done using 8 × 8 matrix, for each character. For better performance of the net, it should be trained for noisy data also. During training, the network was trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural network comes to life when a pattern that has no output associated with it, is given as an input. In this case the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

5.11 Curve Fitting: Neural network is capable of performing curve fitting. It can be trained for some portion of the curve (function) and can be used to predict the future points on the curve. Time series method has been used in this work for curve fitting problems. Typically previous three or four values are used for predicting the next value on the curve, the network is trained in this manner and is then used to predict the future values. It has been found that in case of curves like the sine(or cosine), the network requires to be trained for one complete cycle(2*pi) and it can be used to predict further values. In case of sineexp function, the network needs to capture only a part of the curve, and further predictions can be done by the network.

Chapter 6

Results and Discussions

In the present thesis work, some of problems that have been discussed in the previous chapter (Chapter 5) have been solved. These are basically benchmark problems and require rigorous efforts to find an efficient solution. Most of these problems do not have a direct solution and require searching in a very methodical manner for the ideal solution. Most of these have an optimum solution and the effort has been to find this or something very close to this, since it would be very difficult to locate the global optimum solution. Back-propagation algorithms have been employed, along with its variants. Both first order and second order algorithms have been used for this purpose. The solutions found using MATLAB have been tested on the codes written in JAVA. The codes written in JAVA is relatively new and has got some constraints, however, most of the solutions found using MATLAB are producing satisfactory results. In the following sections are discussed the solutions to the various problems which have been considered in this thesis work.

6.1 The Exclusive-OR Problem

Theoretically, this problem is linearly non-separable and cannot be solved by the use of a single neuron in the output layer. It can be solved by feed-forward neural network having single hidden layer with at least two neurons. Traingd (MATLAB) was the training algorithm used and the activation functions used for the hidden layer and the output layers were logsig and logsig respectively. Logsig is same as unipolar sigmoidal activation function. Beside the theoretical structure various other structures have also been found that give good results in lesser iterations. The results and simulations are shown in figure 6.1 for MATLAB and figures 6.2 and 6.3 for JAVA architectures respectively. The various structures used are listed below in table 6.1.

MATLAB RESULTS

Table No 6.1 :Exclusive-OR

Exclusive-OR: Artificial Neural Network Architectures used; Error Goal = 0.0001					
Network	Hidden Layers	Neurons	Activation Fn	Algorithm	Iterations
XOR1	1	3	Logsig, Tansig	Trainrp	310
XOR2	1	3	Logsig, Logsig	Traingdm	38008
XOR3	1	3	Tansig, Tansig	Traingd	866
XOR4	1	2	Logsig, Logsig	Traingd	33929
XOR5	1	3	Tansig, Logsig	Traingd	9597
XOR8	1	3	Tansig, Logsig	Trainrp	48

Trainrp = Resilient Propagation Algorithm

Traingd = Standard Back-propagation Algorithm

Traingdm = Standard Back-propagation with Momentum

JAVA RESULTS

Table No 6.2 :Exclusive-OR

Exclusive-OR: Artificial Neural Network Architectures used; Error Goal = 0.0001					
Network	Hidden Layers	Neurons	Activation Fn	Algorithm	Iterations
XOR1	1	3	Logsig, Tansig	Trainrp	176
XOR3	1	3	Tansig, Tansig	Traingd	744
XOR4	1	2	Logsig, Logsig	Traingd	2000
XOR8	1	3	Tansig, Logsig	Trainrp	77

Conclusion: From the above results it is concluded that:-

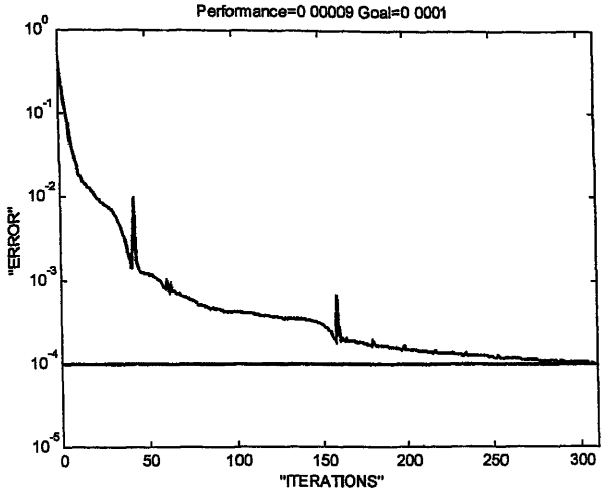
(a) With only two neurons in the hidden layer, it requires large number of iterations to achieve proper classification.

(b) Resilient propagation has given the fastest training, in both MATLAB and JAVA.

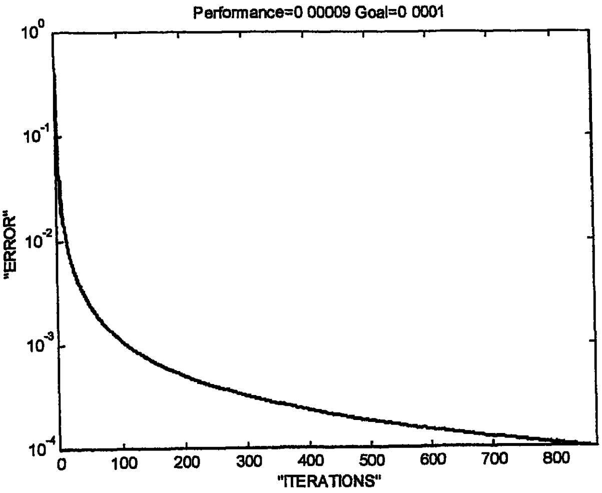
(c) JAVA codes were faster in training.

(d) The theoretical structure gave faster result in the case of JAVA.

XOR1: [3 1] {Logsig Tansig} Trainrp ; Epochs =310



XOR3: [3 1] :{Tansig Tansig}Traingd; Epochs=866



XOR4:[2 1] :{Logsig Tansig} Trainrp; Epochs = 33929

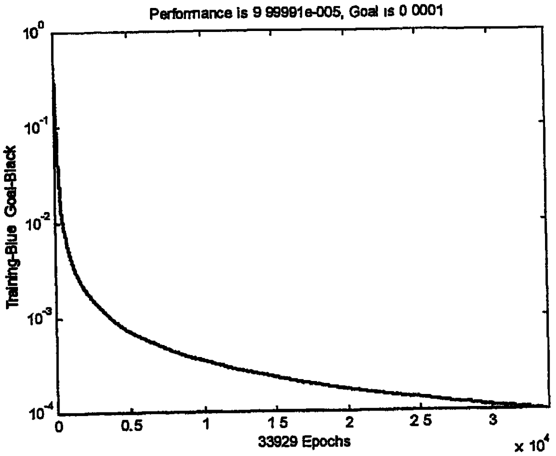
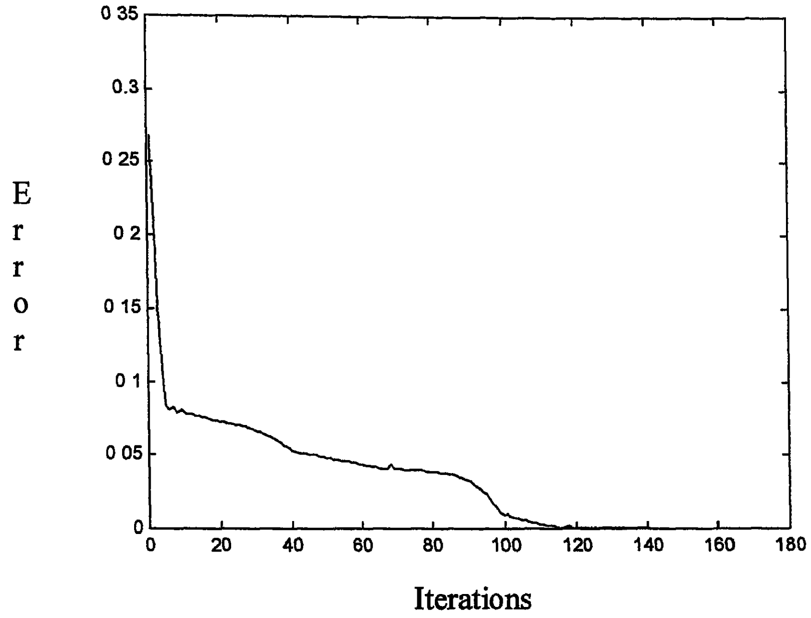


Figure 6. 1 : Training Plots (XOR)

JAVA PLOT

XOR1: [3 1] {logsig tansig} Trainrp ; Epochs =176



XOR3: [3 1] :{tansig tansig}Traingd; Epochs=~~1790~~ 744

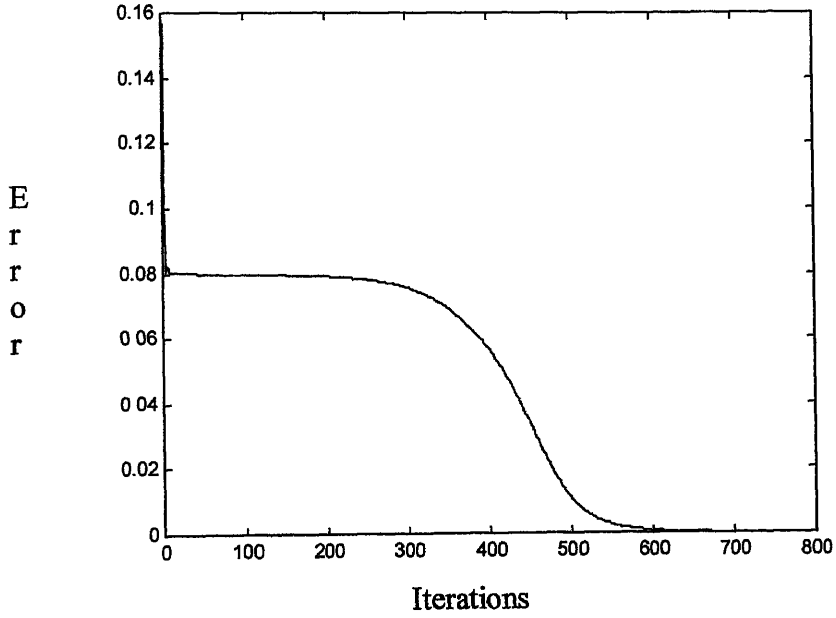
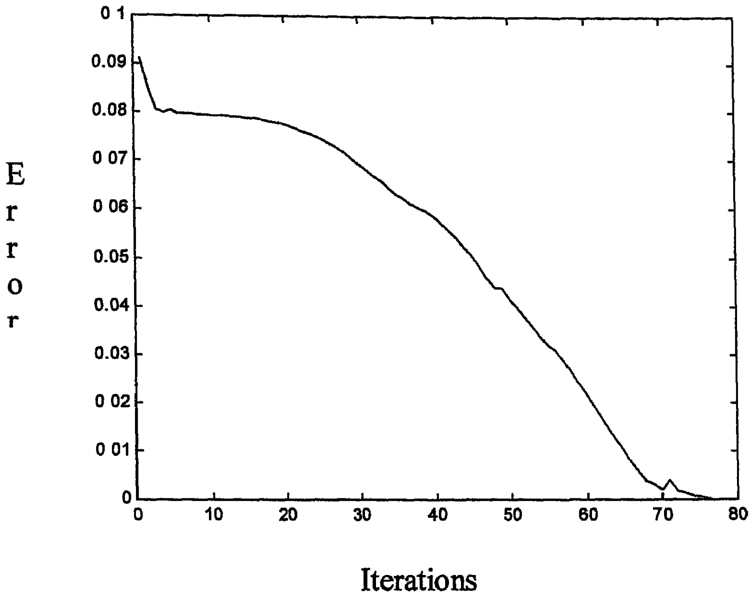


Figure 6.2 : Exclusive-OR Error Plot

JAVA PLOTS

XOR 8: [3 1] {Tansig Logsig } TrainRP; Epochs : 77



XOR 4: [3 1] {Logsig Logsig} TrainRP Epochs: 2000

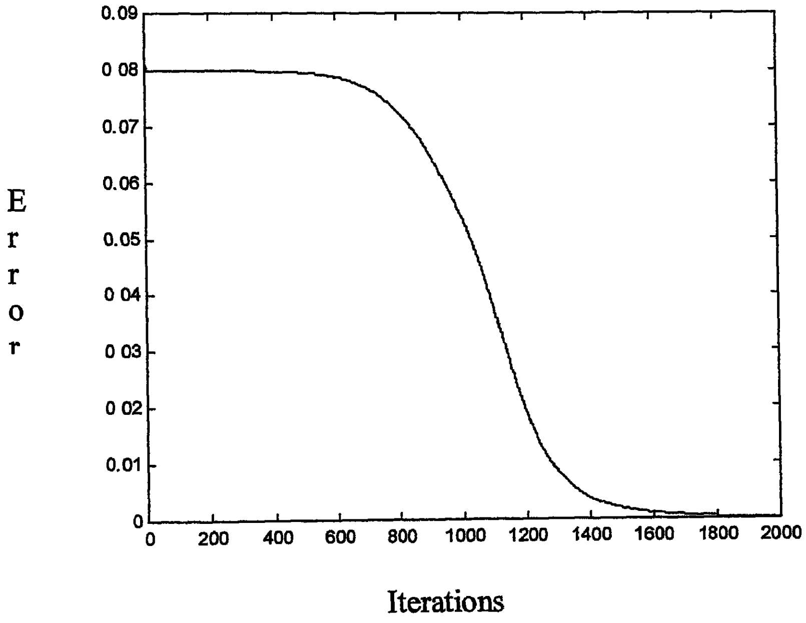


Figure 6.3 : Exclusive-or Error plot

6.2 4-Parity Problem

In this work 4-bit parity problem has been considered. If the input pattern contains an even number of 1^s then its parity is 1 else its 0. This is a difficult problem because the pattern that are closer (using Euclidean distance) in the measure (sample) space, i.e. numbers that differ in only one bit require their answer to be different. The results and simulations are shown in figures 6.4 –6.5 for MATLAB and figures 6.6 to 6.9 for JAVA. For N = 4 the training set is:

Table 6.3:4- Layer Training Set

b3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
b0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Parity	1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1

The general network architecture used for N-Parity problem is shown in figure 5.4. For the present problem various architectures were found that gave good results. These architectures are listed below:

MATLAB RESULTS

Table 6.4: 4-Bit Parity Problem

4-Bit Parity Problem: ANN Architectures used; Error Goal = 0.01				
Network	Architecture	Activation Fn.	Algorithm	Iterations
Parity	[4 4 1]	{Tansig Tansig Tansig}	Traingd	1279
Parity1	[4 4 1]	{Tansig Tansig Tansig}	Trainrp	214
Parity2	[4 4 1]	{Tansig Tansig Logsig}	Traingd	**No Convergenc**
Parity3	[4 4 1]	{Tansig Logsig Logsig}	Traingd	**No Convergenc**
Parity4	[4 4 1]	{Logsig Logsig Tansig}	Traingd	**No Convergenc**
Parity5	[4 4 1]	{Logsig Logsig Logsig}	Trainrp	**No Convergenc**
Parity6	[4 4 1]	{Tansig Logsig Logsig}	Trainlm	31
Parity7	[4 4 1]	{Tansig Tansig Logsig}	Trainscg	168

Parity8	[4 4 1]	{Tansig Logsig Logsig}	Trainscg	**No Convergenc**
Parity9	[4 4 1]	{Logsig Logsig Logsig}	Trainscg	58
Parity10	[4 4 1]	{Tansig Tansig Tansig}	Trainscg	758
Parity11	[4 4 1]	{Tansig Tansig Logsig}	Trainrp	**No Convergenc**

JAVA RESULTS

Table 6.5: 4-Bit Parity Problem

4-Bit Parity Problem: ANN Architectures used; Error Goal = 0.01

Network	Architecture	Activation Fn.	Algorithm	Iterations
Parity	[4 4 1]	{Tansig Tansig Tansig}	Traingd	**No Convergenc**
Parity1	[4 4 1]	{Tansig Tansig Tansig}	Trainrp	261/313
Parity2	[4 4 1]	{Tansig Tansig Logsig}	Traingd	24313/5349
Parity3	[4 4 1]	{Tansig Logsig Logsig}	Traingd	**No Convergenc**
Parity4	[4 4 1]	{Logsig Logsig Tansig}	Traingd	**No Convergenc**
Parity5	[4 4 1]	{Logsig Logsig Logsig}	Trainrp	10000, Error Reached=.016
Parity6	[4 4 1]	{Tansig Logsig Logsig}	Trainlm	17395
Parity7	[4 4 1]	{Tansig Tansig Logsig}	Trainscg	302
Parity8	[4 4 1]	{Tansig Logsig Logsig}	Trainscg	322
Parity9	[4 4 1]	{Logsig Logsig Logsig}	Trainscg	1100
Parity10	[4 4 1]	{Tansig Tansig Tansig}	Trainscg	1572
Parity11	[4 4 1]	{Tansig Tansig Logsig}	Trainrp	1080

Conclusion: From the above results it is concluded that:-

(a) Trainlm (Levenberg-Marquardt Back-propagation) was the fastest algorithm followed by the Scaled Conjugate Gradient Back-propagation, in the case of MATLAB. Whereas, it took lot of iterations (longest) in the case of JAVA.

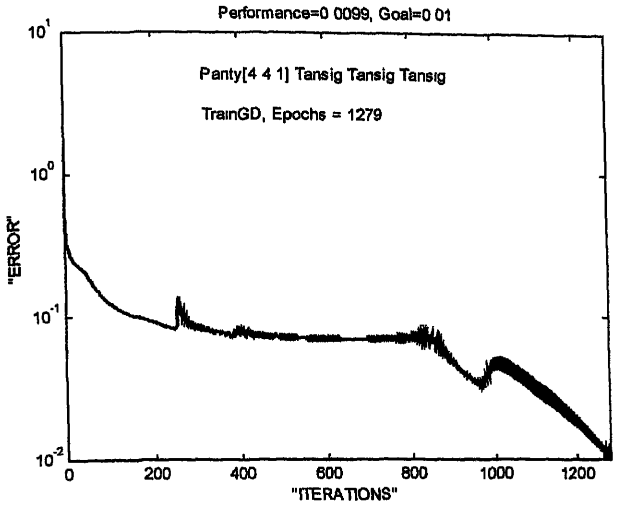
(b) Using unipolar sigmoid activation function (Logsig) gave faster result, compared to bipolar sigmoidal activation function (Tansig).

(c) Resilient works well with most of the architectures in JAVA. It however does not work for most of the architectures in MATLAB.

(d) Standard Back-propagation does not work well with most of the architectures, in both MATLAB and JAVA.

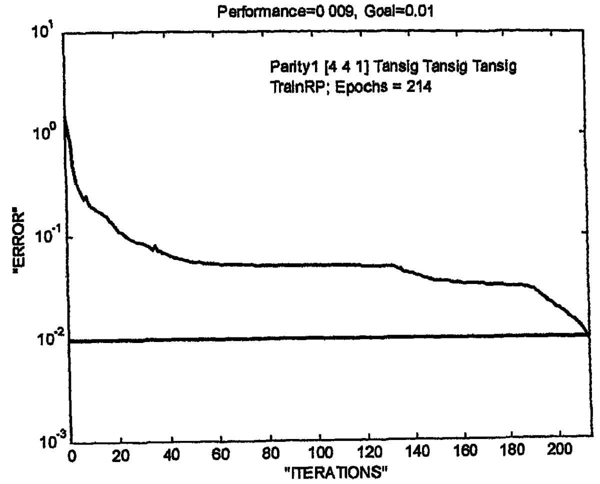
(e) Scaled Conjugate Gradient algorithm works well for almost all network architectures.

Parity **Simulated Output**



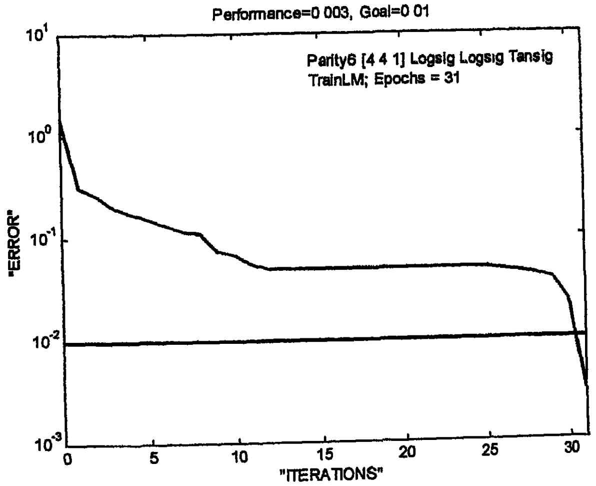
0.9821	1
-0.0286	0
-0.1166	0
0.9932	1
-0.0475	0
0.9498	1
0.9693	1
0.0054	0
-0.1885	0
0.9136	1
0.7761	1
-0.1315	0
0.9703	1
-0.0119	0
-0.1646	0
0.9859	1

Parity 1



0.9479	1
0.0913	0
-0.0264	0
0.9706	1
0.0669	0
0.6574	1
0.9568	1
-0.0820	0
-0.0360	0
0.9749	1
0.9715	1
-0.0140	0
0.9519	1
0.0001	0
0.0196	0
0.9803	1

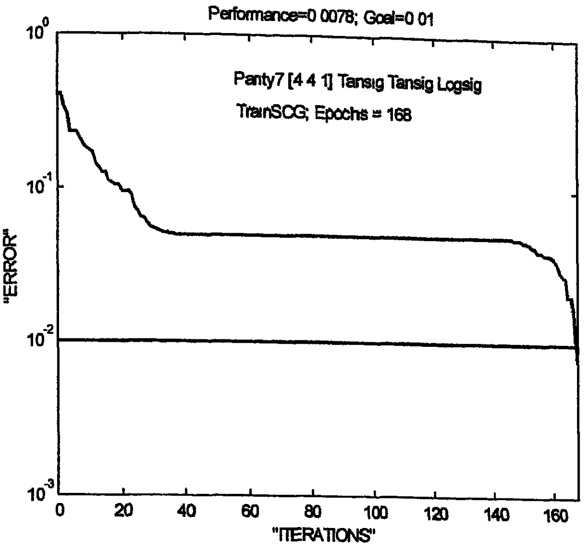
Parity 6



0.9825	1
0.0046	0
-0.0604	0
0.9785	1
0.0139	0
0.9919	1
0.9845	1
0.0081	0
-0.0584	0
0.9784	1
0.8021	1
-0.0106	0
0.9844	1
0.0085	0
-0.0305	0
0.9812	1

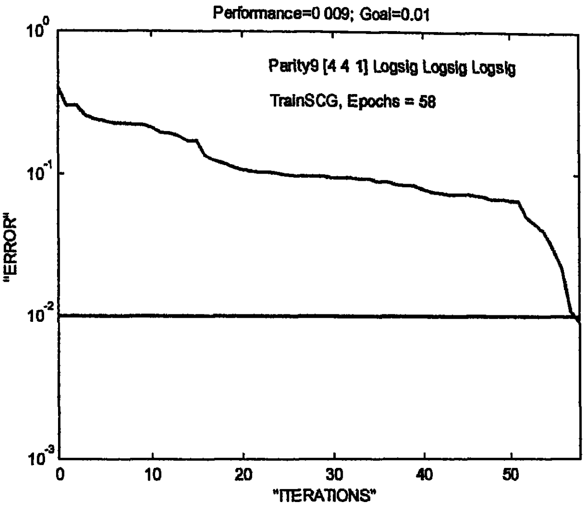
Figure 6.4 : Parity Plots and Simulations (MATLAB)

Parity 7



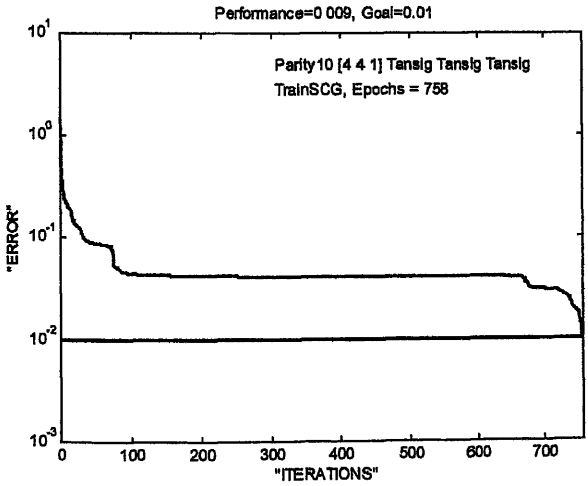
Simulated O/P	Actual O/P
0.9970	1
0.0018	0
0.0921	0
0.8703	1
0.0005	0
0.9848	1
0.9883	1
0.0005	0
0.0774	0
0.9502	1
0.7006	1
0.0306	0
0.9885	1
0.0006	0
0.0367	0
0.9881	1

Parity 9



0.9594	1
0.1086	0
0.1249	0
0.9501	1
0.1099	0
0.8381	1
0.9502	1
0.0610	0
0.0701	0
0.9326	1
0.7873	1
0.0514	0
0.9340	1
0.0973	0
0.0303	0
0.9998	1

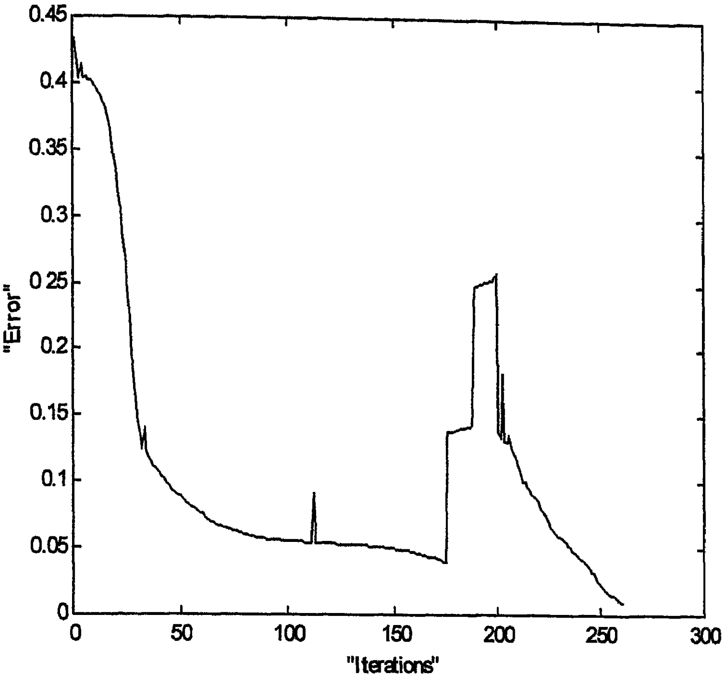
Parity 10



0.9771	1
0.1219	0
0.0218	0
0.9995	1
0.0711	0
0.6496	1
0.9994	1
-0.0787	0
-0.0073	0
0.9995	1
0.9996	1
0.0672	0
0.9988	1
0.0625	0
0.0069	0
0.9999	1

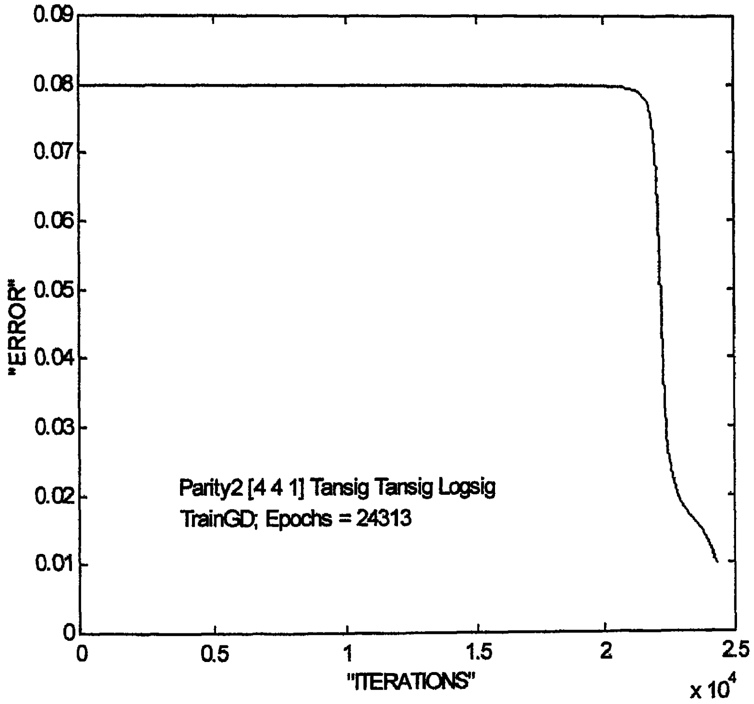
Figure 6.5 :Parity Error Plots and Simulations

Parity 1 [4 4 1] {Tansig Tansig Tansig} Trainrp
Epochs: 261



Simulated O/P	
0.9981	1
0.0017	0
0.0564	0
0.9987	1
0.1021	0
1.0011	1
0.9776	1
0.0089	0
0.0789	0
1.0999	1
0.9657	1
-0.0098	0
0.8767	1
-0.0098	0
-0.0987	0
1.1099	1

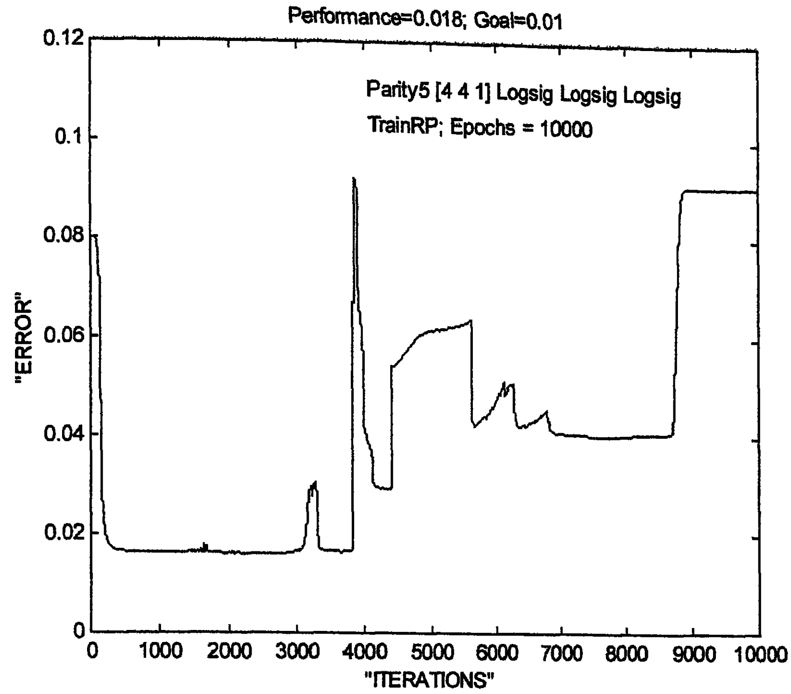
Parity 2 [4 4 1] {Tansig Tansig Logsig} Traingd
Epochs: 24313



0.9981	1
0.0017	0
0.0564	0
0.9987	1
0.1021	0
1.0011	1
0.9776	1
0.0089	0
0.0789	0
1.0999	1
0.9657	1
-0.0098	0
0.8767	1
-0.0098	0
-0.0987	0
1.1099	1

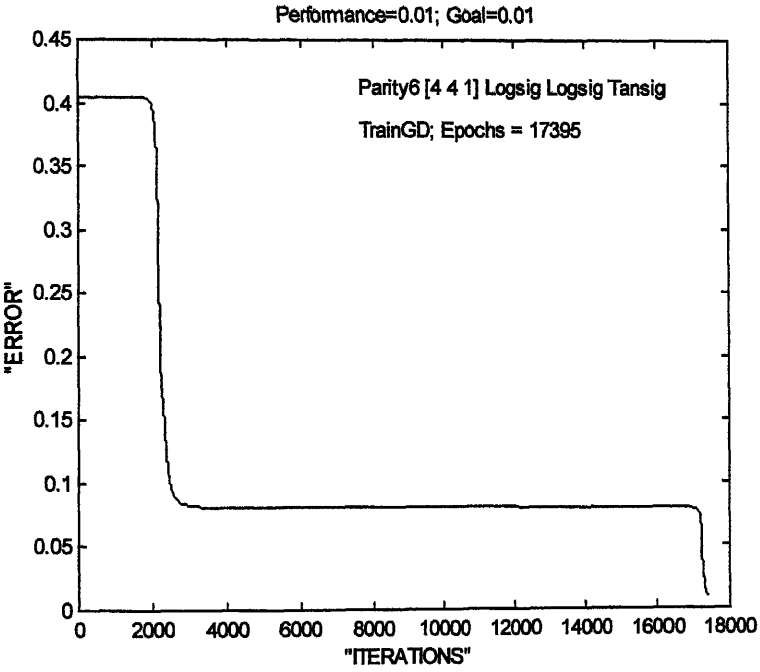
Figure 6. 6 : Parity Plot

F



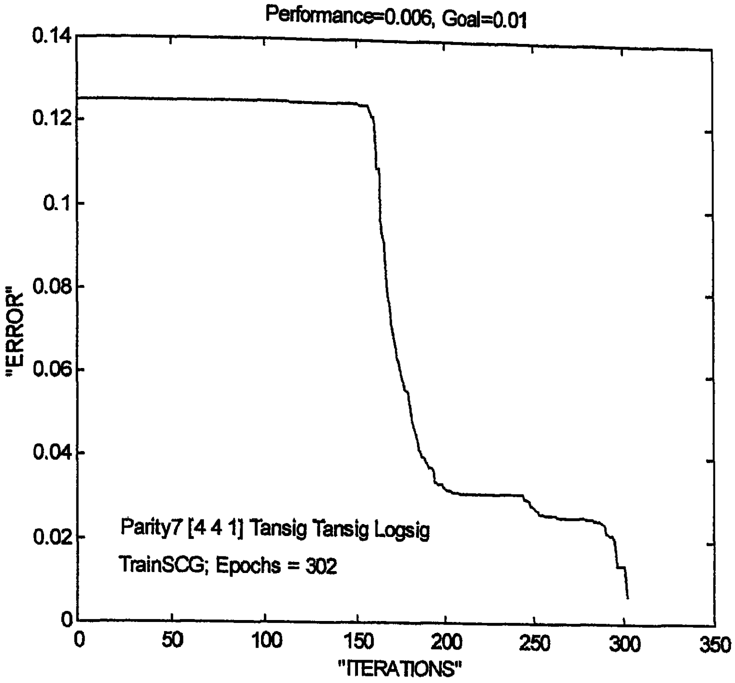
Simulated O/P	
0.9681	1
0.0617	0
0.0764	0
0.6087	1
0.1021	0
1.0011	1
0.9776	1
0.1089	0
0.0789	0
1.0999	1
0.5657	1
-0.0098	0
0.8767	1
-0.4098	0
0.4987	0
0.4099	1

Parity 6



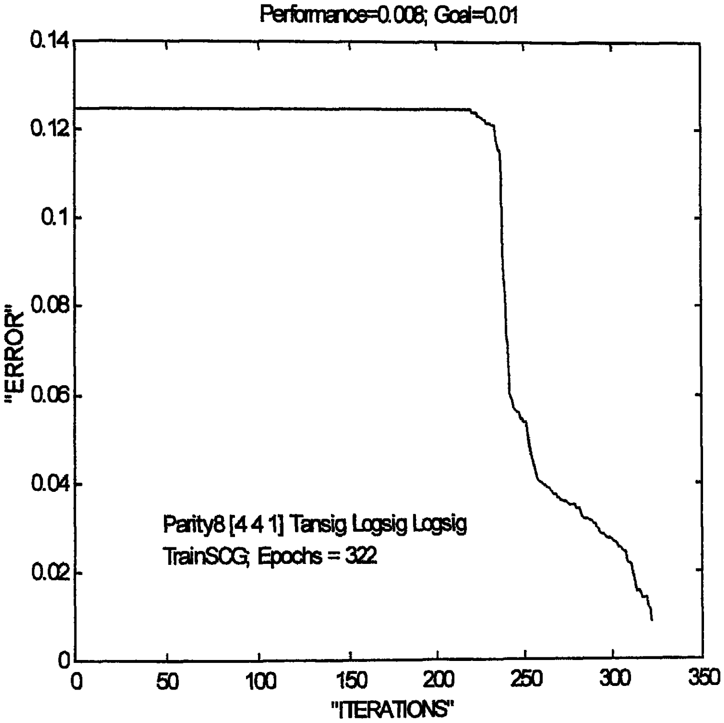
0.9681	1
0.0617	0
0.0764	0
1.0087	1
0.1021	0
1.0011	1
0.9776	1
0.1089	0
0.0789	0
1.0999	1
0.9657	1
-0.0098	0
0.9767	1
-0.0098	0
0.0987	0
0.9899	1

Figure 6. 7: Parity Error Plots



Simulated O/P	
1.0000	1
0.2122	0
0.0383	0
1.0000	1
0.0607	0
1.0000	1
0.9808	1
0.0085	0
0.1373	0
1.0000	1
0.9322	1
0.0191	0
0.9687	1
0.0929	0
0.3507	0
0.9205	1

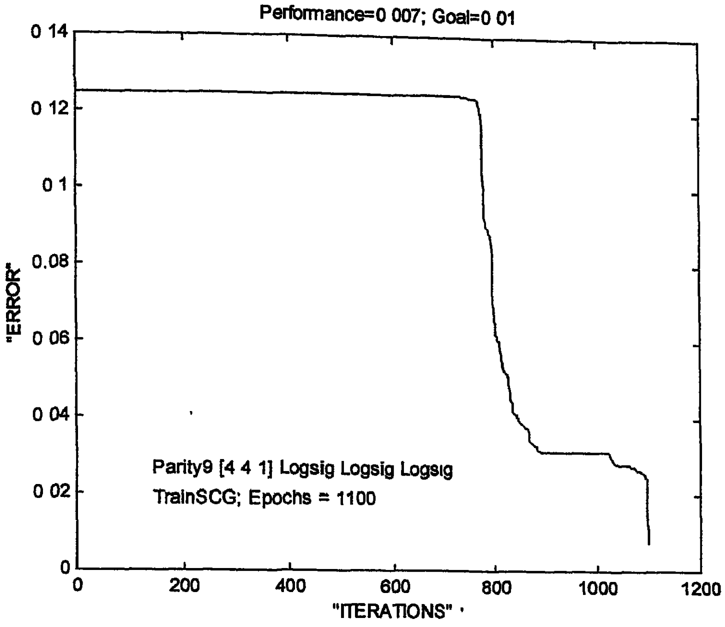
Parity 8



0.8757	1
0.0036	0
0.0047	0
0.9886	1
0.0006	0
0.9711	1
0.9853	1
0.0321	0
0.4003	0
0.8610	1
0.7735	1
0.0171	0
0.8185	1
0.0077	0
0.0356	0
0.9402	1

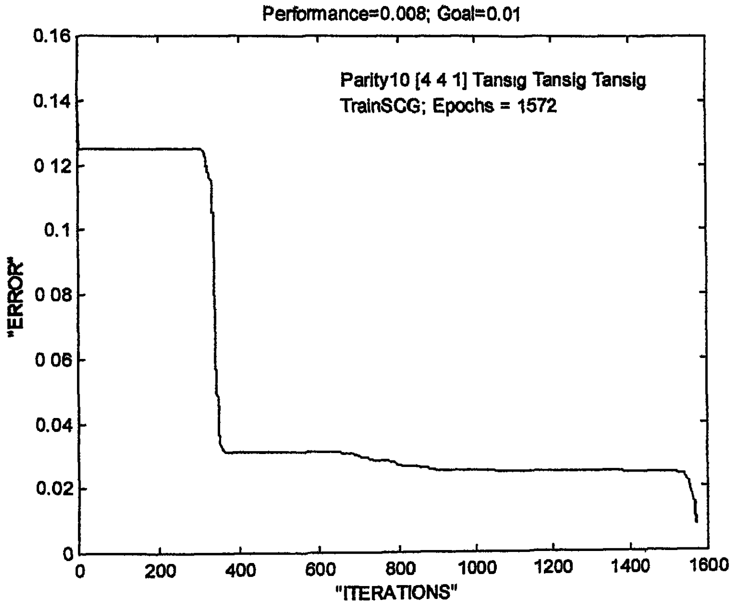
Figure 6. 8: Parity Plots and Simulations

Parity 9



Simulated O/P	
0.7480	1
0.0051	0
0.3978	0
0.9164	1
0.0040	0
0.9979	1
0.9596	1
0.0037	0
0.0219	0
0.9978	1
0.9299	1
0.0062	0
0.9945	1
0.0000	0
0.0032	0
0.9945	1

Parity 10



0.8033	1
-0.0286	0
0.0108	0
0.7959	1
0.0326	0
0.8681	1
0.7676	1
0.0195	0
-0.0314	0
0.8679	1
0.8031	1
-0.0506	0
0.8891	1
0.1807	0
0.0128	0
0.8652	1

Figure 6.9: Parity Error Plots and Simulations

6.3 CODEC Problem

The codec problem is a typical identity-mapping problem. Here 16-bit input code has been mapped to its own self, i.e. 16-bit output code. The network is trained to produce an output binary pattern, which is identical to the input pattern. This forces the hidden units to encode the patterns. For example, with n =16 and m = 4, the network must provide an equivalent binary log_2n encoding in the hidden layer. When m is small compared to n, the task becomes more difficult to learn than when m is near n. This is, sometimes, referred to as *tight encoding*. This benchmark problem is important to evaluate the network storage capacity, and it has application to data compression and transmission. A total of 16 codes have been used for the purpose of training. Theoretically this problem can be solved by using a single hidden layer with the number of neurons being equal to $\log_2$N, where N is the number of input. Thus 4 neurons have been used in the hidden layer. Two architectures have been found that give good results with the restraints of the hidden layer and the number of neurons. These have been tested for both MATLAB as well as JAVA. The error convergence plot and the results are shown in figure 6.10 for MATLAB and figures 6.11 – 6.12 for JAVA. The two architectures that were found to give proper convergence are:

MATLAB RESULTS

Table 6.6: CODEC Problem

CODEC Problem: ANN Architectures used; Error Goal = 0.0001				
Network	Architecture	Activation Fn.	Algorithm	Iterations
Codec	[4 16]	{Logsig Logsig}	Trainrp	74
Codec1	[4 16]	{Tansig Logsig}	Trainrp	85
Codec3	[4 16]	{Tansig Logsig}	Trainscg	610

JAVA RESULTS

Table 6.7: CODEC Problem

CODEC Problem: ANN Architectures used; Error Goal = 0.0001				
Network	Architecture	Activation Fn.	Algorithm	Iterations
Codec	[4 16]	{Logsig Logsig}	Trainrp	95/96
Codec1	[4 16]	{Tansig Logsig}	Trainrp	86/203
Codec3	[4 16]	{Tansig Logsig}	Trainrp	>10000

Conclusion: From the results obtained it can be concluded that:

(a) Using unipolar sigmoidal activation function and Resilient Back-propagation gives fast convergence and good simulation in case of MATLAB.

(b) Using the same architecture in JAVA took comparatively a very long time for error convergence.

(c) ANN built using JAVA requires less rigorous training as far as error goal is concerned.

TrainSCG worked for MATLAB, but it did not show good convergence for JAVA.

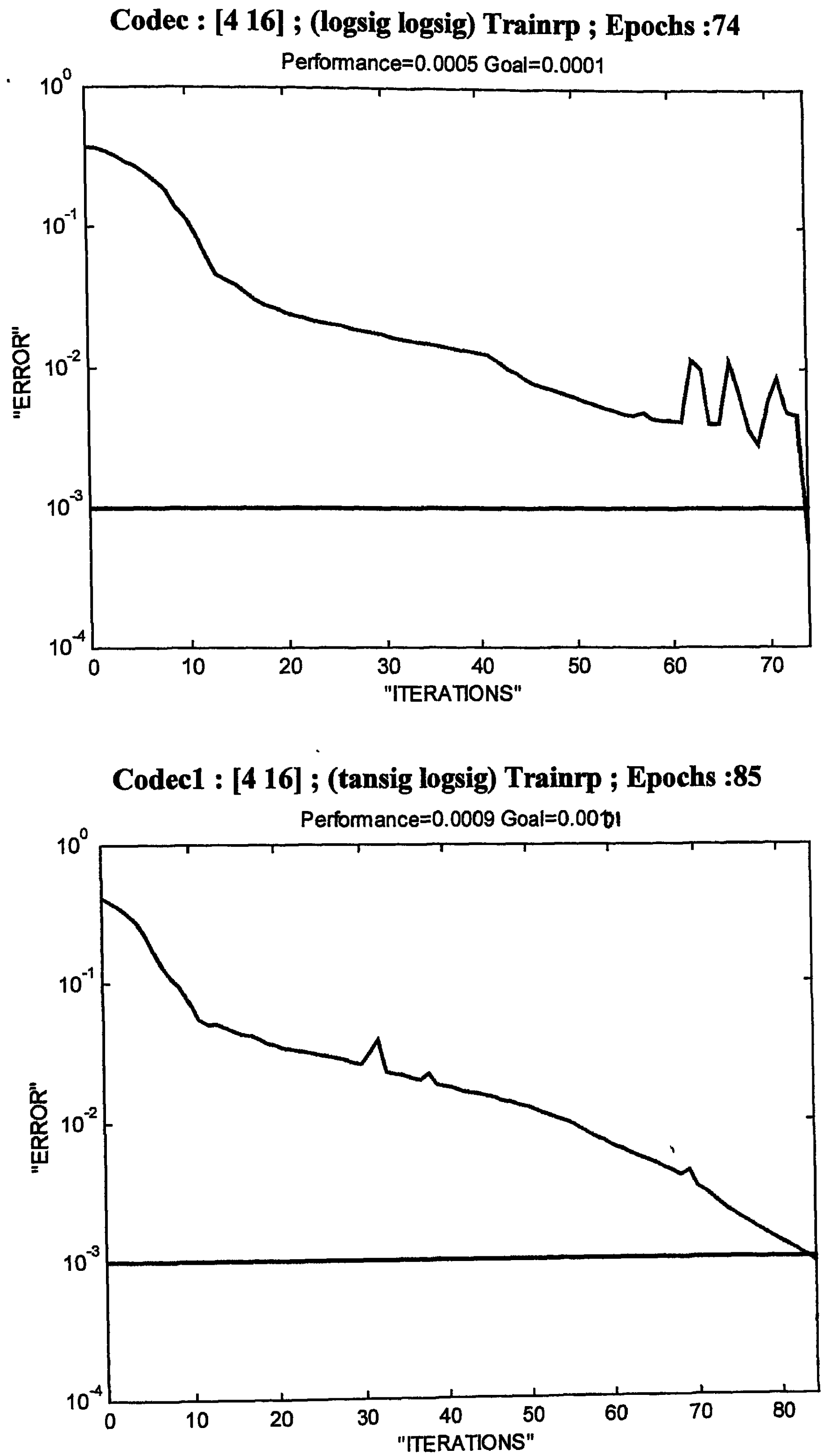


Figure 6.10: CODEC Error Plot

Simulated Output of Codec

```
                          Columns 1 through 7

  0.0018    0.0000    0.0000         0         0         0    0.0057
  0.0046    0.0000    0.0016    0.0000    0.0044         0    0.0001
  0.0000    0.0000    0.0000    0.0000    0.0000         0    0.0034
  0.0000    0.0000    0.0000    0.0000    0.0000         0    0.0000
  0.0000    0.0008    0.0000    0.0000    0.0000    0.0009    0.0123
  0.0110    0.0000    0.0023    0.0000    0.0000    0.0000    0.0145
  0.0008    0.0002    0.0039    0.0000         0    0.0011    0.0063
  0.0000    0.0635    0.0000         0         0    0.0000    0.0013
  0.0000    0.0022    0.0062    0.0035    0.0000    0.0062    0.0007
  0.0000    0.0000    0.0000         0         0    0.0000    0.9857
  0.0000    0.0814    0.0020    0.0000    0.0000    0.9764    0.0000
  0.0000    0.0005    0.0000    0.0019    0.9861    0.0000    0.0006
  0.0000    0.2237    0.0002    0.9886    0.0381    0.0000    0.0000
  0.0052    0.0005    0.9872    0.0000    0.0000    0.0000    0.0001
  0.0000    0.7365    0.0000    0.0000    0.0000    0.0088    0.0000
  0.9826    0.0135    0.0050    0.0000         0    0.0000    0.0000

                          Columns 8 through 14

  0.0000    0.0000         0    0.0000    0.0000    0.0000    0.0001
  0.0000    0.0000         0    0.0000    0.0000    0.0248    0.0068
  0.0000    0.0000         0    0.0293    0.0000    0.0128    0.9726
  0.0000    0.0000         0         0    0.0000    0.9719    0.0000
  0.0014    0.0000    0.0000    0.0010    0.9836    0.0001    0.0205
  0.0000    0.0000         0    0.9969    0.0000    0.0003    0.0351
  0.0000    0.0019    0.9948    0.0000    0.0000    0.0000    0.0000
  0.0000    0.9765    0.0000         0    0.0000    0.0000    0.0000
  0.9907    0.0000    0.0000    0.0000    0.0043    0.0005    0.0017
  0.0000    0.0000    0.0000    0.0000    0.0014    0.0000    0.0109
  0.0023    0.0015    0.0061    0.0000    0.0000    0.0080    0.0000
  0.0000    0.0000    0.0000    0.0000    0.0002    0.0012    0.0104
  0.0092    0.0004    0.0000    0.0000    0.0000    0.0247    0.0000
  0.0022    0.0000    0.0000    0.0000    0.0000    0.0014    0.0000
  0.0000    0.0312    0.0000         0    0.0000    0.0315    0.0000
  0.0000    0.0000    0.0000         0    0.0000    0.0049    0.0000

                          Columns 15 through 16

  0.0000    0.9762
  0.9842    0.0159
  0.0002    0.0018
  0.0084    0.0081
  0.0000    0.0000
  0.0000    0.0143
  0.0000    0.0007
  0.0000    0.0007
  0.0060    0.0000
  0.0000    0.0105
  0.0000    0.0000
  0.0157    0.0000
  0.0011    0.0000
  0.0059    0.0003
  0.0000    0.0000
  0.0000    0.0277
```

Figure 6.10b: Codec Simulated

Codec1 Simulated Values

```
                         Columns 1 through 7

0.0000    0.0017    0.0000    0.0015    0.0024    0.0661    0.0000
0.0000    0.0015    0.0000    0.0303    0.0000    0.0001    0.0050
0.0000    0.0907    0.0000    0.0125    0.0111    0.0001    0.0000
0.0576    0.0006    0.0000    0.0294    0.0000    0.0008    0.0643
0.0266    0.0458    0.0000    0.0000         0    0.0029    0.0551
0.0000    0.0569    0.0098    0.0000    0.0000    0.0003    0.0000
0.0000    0.0105    0.0189    0.0000         0    0.0503    0.0000
0.0000    0.0006    0.0000    0.0054    0.0000    0.0391    0.0000
0.0000    0.0708    0.0000    0.0000         0    0.0010    0.0090
0.0000    0.0570    0.0000    0.0084    0.0000    0.0000    0.9277
0.0000    0.0220    0.0000    0.0000         0    0.9273    0.0000
0.0000    0.0079    0.0000    0.0002    0.9716    0.0035    0.0000
0.0000    0.0029    0.0000    0.9548    0.0000    0.0003    0.0002
0.0000    0.0238    0.9934    0.0000    0.0000    0.0082    0.0000
0.0000    0.8542    0.0000    0.0000         0    0.0019    0.0000
0.9696    0.0415    0.0000    0.0001         0    0.0074    0.0132

                         Columns 8 through 14

0.0000    0.0187    0.0000    0.0000         0    0.0000    0.0267
0.0113    0.0019    0.0004    0.0403    0.0000    0.0036    0.0376
0.0000    0.0000    0.0000    0.0003         0    0.0000    0.8681
0.0016    0.1264    0.0271    0.0001    0.0001    0.7984    0.0004
0.0004    0.0019    0.0000    0.0000    0.9788    0.0401    0.0000
0.0000    0.0000    0.0000    0.9739         0    0.0000    0.0147
0.0185    0.0226    0.9877    0.0000    0.0000    0.0007    0.0000
0.0000    0.9046    0.0000    0.0000    0.0000    0.0538    0.0000
0.9719    0.0000    0.0209    0.0020    0.0000    0.0002    0.0000
0.0003    0.0000    0.0000    0.0007    0.0072    0.0048    0.0001
0.0000    0.0283    0.0041    0.0000         0    0.0000    0.0000
0.0000    0.0000    0.0000    0.0033         0    0.0000    0.1403
0.0000    0.0630    0.0000    0.0000    0.0000    0.0018    0.0290
0.0000    0.0000    0.0116    0.0337         0    0.0000    0.0000
0.0000    0.0000    0.0000    0.0000         0    0.0000    0.0000
0.0000    0.0378    0.0000    0.0000    0.0211    0.1053    0.0000

                         Columns 15 through 16

0.0035    0.9931
0.9260    0.0236
0.0003    0.0000
0.0236    0.0023
0.0000    0.0000
0.0004    0.0000
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0239    0.0736
0.0003    0.0002
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
```

Figure 6.10c: Simulated Values of Codec1

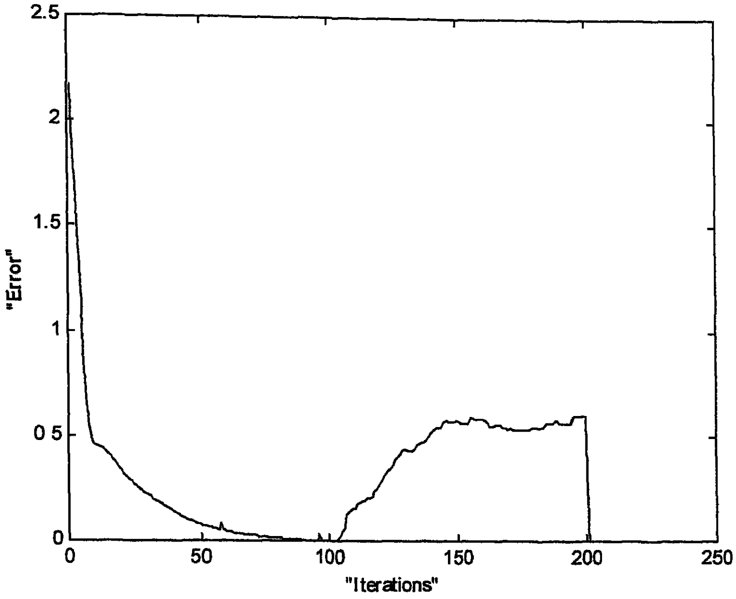
Simulated Values for Codec3

```
                         Columns 1 through 7

  0.0000    0.0000    0.0000    0.1246    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0029    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0427    0.0121    0.0000    0.0000    0.0000
  0.0051    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0007    0.0095    0.0000    0.0000    0.0000    0.0018    0.0000
  0.0000    0.0098    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0022    0.0493    0.0080    0.0000
  0.0106    0.0000    0.0000    0.0000    0.0000    0.0000    0.0118
  0.0000    0.0000    0.0950    0.0240    0.0000    0.0100    0.9802
  0.0000    0.0000    0.0000    0.0032    0.0074    0.9317    0.0152
  0.0000    0.0000    0.0000    0.1463    0.8843    0.0514    0.0000
  0.0000    0.0000    0.0685    0.7390    0.0943    0.0000    0.0000
  0.0000    0.0000    0.8957    0.1794    0.0000    0.0000    0.0047
  0.0058    0.9634    0.0000    0.0000    0.0000    0.0000    0.0000
  0.9774    0.0020    0.0000    0.0000    0.0000    0.0000    0.0000

                         Columns 8 through 14
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0353
  0.0000    0.0000    0.0034    0.0000    0.0000    0.0055    0.0000
  0.0000    0.0006    0.0390    0.0000    0.0000    0.0000    0.9405
  0.0000    0.0000    0.0000    0.0000    0.0191    0.9767    0.0000
  0.0001    0.0000    0.0000    0.0000    0.9818    0.0021    0.0000
  0.0000    0.0162    0.0000    0.9826    0.0000    0.0000    0.0000
  0.0000    0.0000    0.9768    0.0000    0.0000    0.0000    0.0382
  0.0000    0.9287    0.0000    0.0130    0.0000    0.0000    0.0006
  0.9701    0.0000    0.0000    0.0000    0.0061    0.0000    0.0000
  0.0547    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0094    0.0001    0.0000    0.0229    0.0000    0.0000    0.0000
  0.0000    0.0839    0.0000    0.0000    0.0000    0.0000    0.0001
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0027    0.0000
  0.0000    0.0000    0.0261    0.0015    0.0000    0.0000    0.0000
  0.0071    0.0000    0.0000    0.0002    0.0003    0.0000    0.0000

                         Columns 15 through 16

  0.0000    0.9246
  0.9749    0.0003
  0.0000    0.0386
  0.0124    0.0423
  0.0125    0.0000
  0.0000    0.0000
  0.0060    0.0000
  0.0000    0.0000
  0.0000    0.0000
  0.0000    0.0000
  0.0000    0.0000
  0.0000    0.0000
  0.0000    0.0976
  0.0000    0.0000
  0.0000    0.0000
  0.0000    0.0000
```

Figure 6. 10d: Codec Simulated Output

JAVA PLOTS

Codec [4 16] {Logsig Logsig} Trainrp Epochs: 96



Codec 1 [4 16] {Tansig Logsig} Trainrp Epochs: 203

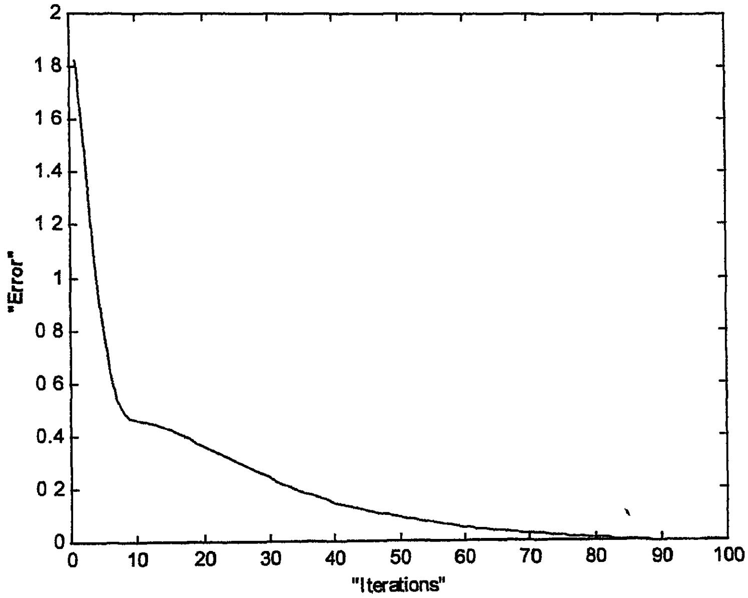
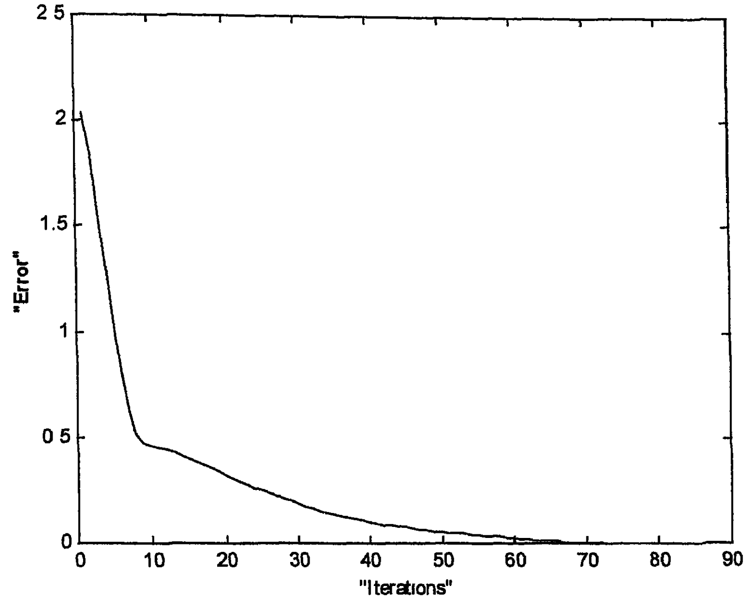


Figure 6.11 : CODEC Error Plots

JAVA PLOTS

Codec: [4 16] {Logsig Logsig} Trainrp Epochs : 95



Codec 1:[4 16] {Tansig Logsig} Trainrp Epochs : 86

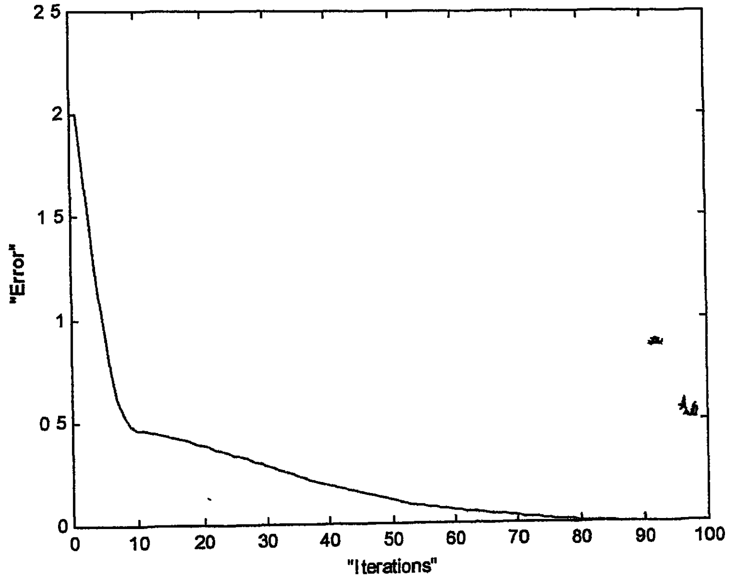


Figure 6.12: CODEC Error Plots

Codec(JAVA) Simulated Values

```
                         Columns 1 through 7

  0.0000    0.0000    0.0000         0    0.0193    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0223    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0019
  0.0000    0.0000    0.0000    0.0000    0.0067    0.0000    0.0000
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0005    0.0000
  0.0000    0.0000    0.0015    0.0000    0.0000    0.0000    0.0010
  0.0000    0.0071    0.0259    0.0000    0.0002    0.0191    0.0000
  0.0000    0.0000    0.0000    0.0219    0.0000    0.0000    0.0000
  0.0001    0.0000    0.0000    0.0000    0.0000    0.0000    0.9999
  0.0000    0.0000    0.0000    0.0000    0.0000    0.9963    0.0000
  0.0000    0.0000    0.0000    0.0000    0.9554    0.0000    0.0000
  0.0000    0.0000    0.0000    0.9820    0.0000    0.0000    0.0000
  0.0000    0.0000    0.9766         0    0.0001    0.0000    0.0003
  0.0000    0.9912    0.0000    0.0000    0.0000    0.0006    0.0000
  0.9999    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000

                         Columns 8 through 14

  0.0000    0.0000    0.0000    0.0000    0.0063    0.0000    0.0000
  0.0171    0.0000    0.0173    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0326    0.0000    0.0000    0.0000    0.0000    0.9984
  0.0000    0.0000    0.0000    0.0000    0.0000    0.9660    0.0000
  0.0000    0.0000    0.0076    0.0000    0.9926    0.0000    0.0000
  0.0000    0.0155    0.0000    0.9983    0.0000    0.0000    0.0007
  0.0192    0.0000    0.9775    0.0000    0.0031    0.0000    0.0000
  0.0017    0.9240    0.0146    0.0009    0.0000    0.0000    0.0022
  0.9504    0.0260    0.0020    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0000    0.0006    0.0000    0.0000    0.0001    0.0000
  0.0000    0.0006    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0206    0.0000    0.0000    0.0004    0.0000    0.0000
  0.0509    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0015    0.0000    0.0000    0.0000    0.0000    0.0000
  0.0000    0.0029    0.0000    0.0000    0.0000    0.0002    0.0000
  0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000

                         Columns 15 through 16

 0.0000    0.9928
 0.9669    0.0000
 0.0000    0.0000
 0         0.0000
 0.0000    0.0054
 0.0000    0.0000
 0.0002    0.0000
 0.0000    0.0000
 0.0019    0.0000
 0.0000    0.0000
 0.0000    0.0000
 0.0000    0.0001
 0.0375    0.0000
 0.0000    0.0049
      0    0.0000
 0.0000    0.0000
```

Figure 6. 12a: Codec Simulated Values

Codec(JAVA) Simulated Values

```
                          Columns 1 through 7

0.0000    0.0000    0.0000    0.0000    0.0000    0.0445    0.0795
0.0025    0.0000    0.0000    0.0000    0.0000    0.0010    0.0012
0         0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0         0.0069    0.0059    0.0000    0.0000    0.0000    0.0310
0.0000    0.0000    0.0000    0.0000    0.0000    0.0059    0.0000
0.0000    0.0000    0.0000    0.0000    0.0000    0.0001    0.0063
0         0.0000    0.0027    0.0000    0.0000    0.0000    0.0152
0         0.0000    0.0079    0.0016    0.0000    0.0000    0.0007
0.0000    0.0000    0.0006    0.0000    0.0024    0.0003    0.0000
0         0.0000    0.0000    0.0000    0.0000    0.0005    0.9268
0.0003    0.0000    0.0000    0.0000    0.0000    0.9602    0.0007
0.0000    0.0000    0.0000    0.0024    0.9587    0.0122    0.0000
0.0000    0.0000    0.0000    0.9913    0.0441    0.0126    0.0000
0         0.0000    0.9923    0.0000    0.0000    0.0000    0.0070
0         0.9917    0.0000    0.0054    0.0000    0.0015    0.0181
0.9957    0.0000    0.0000    0.0000    0.0000    0.0192    0.0009

                          Columns 8 through 14

0.0000    0.0000    0.0000    0.0005    0.0000    0.0000    0.0000
0.0033    0.0000    0.0001    0.0042    0.0126    0.0000    0.0000
0.0000    0.0062    0.0074    0.0000    0.0092    0.0000    0.9745
0.0000    0.0052    0.0000    0.0002    0.0000    0.9982    0.0000
0.0015    0.0000    0.0000    0.0000    0.9821    0.0000    0.0123
0.0000    0.0000    0.0123    0.9924    0.0007    0.0000    0.0000
0.0000    0.0000    0.9824    0.0002    0.0002    0.0000    0.0126
0.0000    0.9867    0.0001    0.0000    0.0000    0.0000    0.0095
0.9948    0.0010    0.0004    0.0000    0.0122    0.0000    0.0072
0.0000    0.0000    0.0000    0.0063    0.0000    0.0000    0.0000
0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000    0.0000    0.0000    0.0027    0.0000    0.0028
0.0000    0.0054    0.0000    0.0000    0.0000    0.0000    0.0001
0.0000    0.0030    0.0117    0.0009    0.0000    0.0011    0.0012
0.0000    0.0001    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000    0.0000    0.0000    0.0039    0.0000    0.0000

                          Columns 15 through 16

0.0000    0.9917
0.9626    0.0019
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0260    0.0154
0.0000    0.0000
0.0000    0.0000
0.0061    0.0000
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0000    0.0000
0.0195    0.0000
```

Figure 6.12b: Codec(JAVA) Simulated Output

6.4 Character Recognition Problem

A popular type of benchmark test used for neural networks is that of character recognition. The number of inputs and outputs will vary depending on the set of characters used for recognition and the choice of features selected for input. In this work the alphabets (A...Z) have been used for recognition. In the most difficult case cursory handwritten character recognition can be attempted. For these tests, the inputs may be individual pixels, say 8 X 8 presented to the network in a particular fashion. The outputs are the binary representations of the individual characters.

In this problem, all the 26 letters have been mapped to 5-bit codes. Each letter has been represented by a 64-bit word written using the 8 X 8 pixels. These bits have been converted to 64-bit. The words have been read in two different fashions. The method of reading affects the performance of the network. The results and the plots are shown in figures 6.13 to 6.16 (MATLAB) and figures 6.17 to 6.19 (JAVA). The architectures found to give good recognition capabilities are as shown in the table below:

MATLAB RESULTS

Table 6.8: Character Recognition Problem

Character Recognition Problem: ANN Architectures used;					
Network	Architecture	Activation Fn.	Algorithm	Error Goal	Iterations
Charalpha_11	[10 11 5]	Tansig Tansig Logsig	Trainrp	0.0001	100
Charalpha_12	[10 11 5]	Tansig Tansig Logsig	Trainrp	0.001	80
Charalpha_21	[10 11 5]	Logsig Logsig Logsig	Trainrp	0.00001	103

JAVA RESULTS

Table 6.9: Character Recognition Problem:

Character Recognition Problem: ANN Architectures used;					
Network	Architecture	Activation Fn.	Algorithm	Error Goal	Epochs
Charalpha	[10 11 5]	Tansig Tansig Logsig	Trainrp	0.00001	100
CharalphaCRO1	[10 11 5]	Tansig Tansig Logsig	Trainrp	0.00001	61
CharalphaCRO2	[10 11 5]	Logsig Logsig Logsig	Trainrp	0.00001	2206

In the case of first architecture of JAVA the input data has been read in a different fashion (Similar to raster scan, from top to bottom), in the rest of the cases the input data has been obtained by reading the pixels similar to CRO scan from top to bottom. In the case of MATLAB reading the alphabet in the first fashion (Raster scan) resulted in inferior recognition of noisy input. The network were trained and tested for noisy patterns and were found to give good recognition capability for the above architectures.

Conclusion:

(a) The number of hidden layers and units vary depending on the application, but typically, no more than two hidden layers are required.

(b) For better performance of the net, it should be trained for noisy data also.

(c) The power of neural network comes to life when a pattern that has no output associated with it, is given as an input. In this case the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

(d) The way the pixels are read affects the noise tolerance of the network.

(e) The performance of JAVA codes was better as far as noisy input was concerned.

(f) Training the network for a very stringy error tolerance causes over-fitting and the network leaves no room for recognition of distorted input.

(g) Reading the patterns top-down and down-top (CRO scan) gives better results.

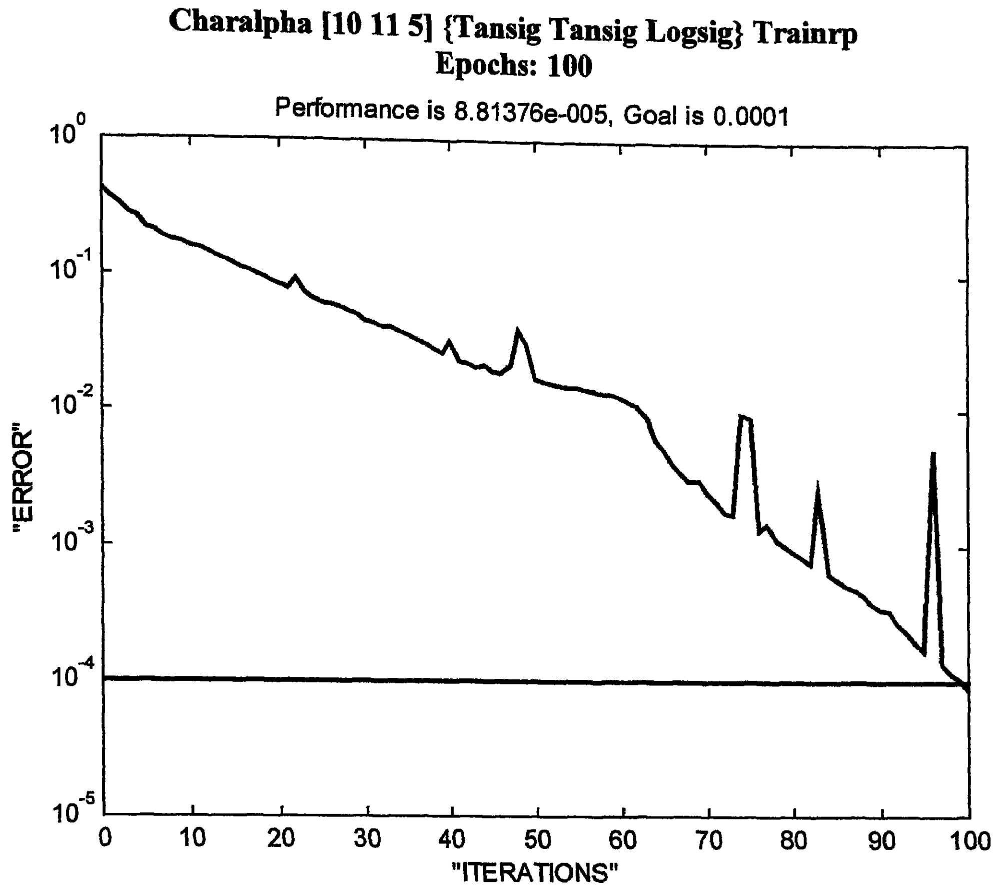
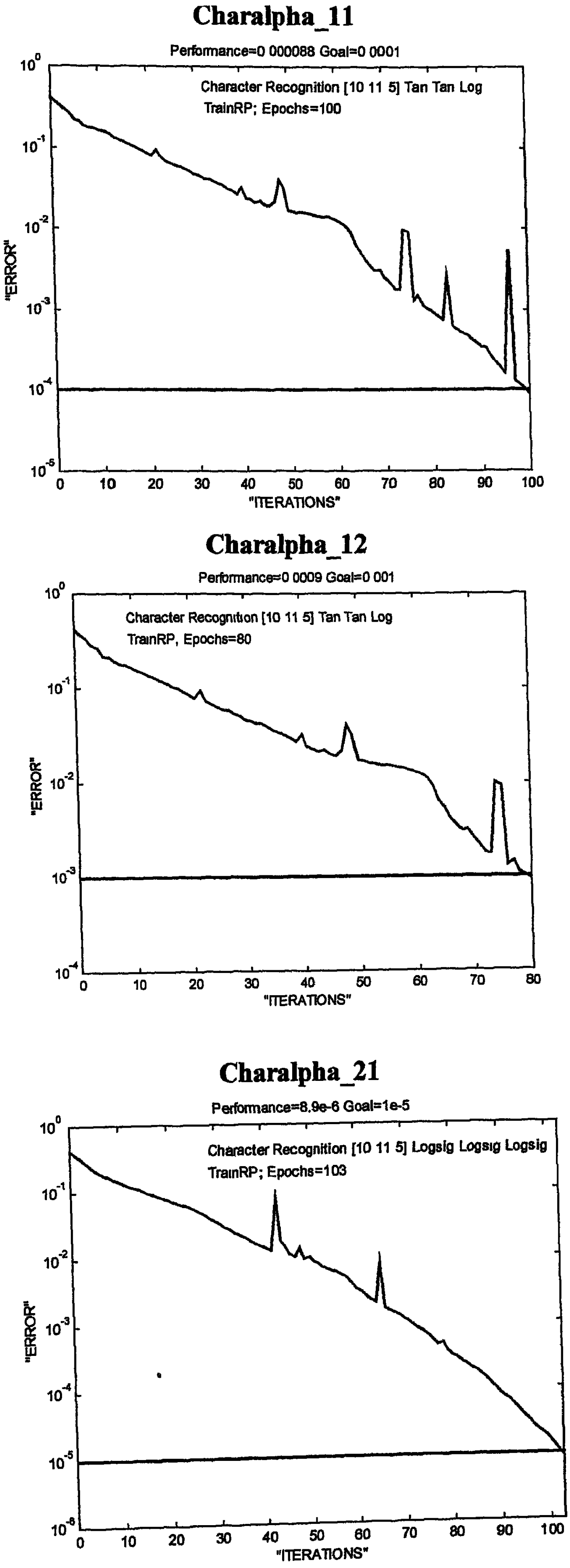


Figure 6.13 : Character Recognition (Alphabets)
(The network was tested for distorted and noisy data
And was found to have good recognition capabilities)



Charalpha_11

Charalpha_12

Charalpha_21

Figure 6.14 : Character Recognition Error Plots

```
Charalpha_11 Simulated Output with noisy input(*=noise)
    0.0000     0.0010     0.0000     0.0012     0.0015
    0.0008     0.0067     0.0008     0.0226     0.9748
    0.0060     0.0020     0.0000     1.0000     0.0001
    0.0042     0.0028     0.0000     0.9999     0.9937
    0.0001     0.0017     1.0000     0.0008     0.2378*
    0.0000     0.0128     1.0000     0.0030     0.9977
    0.0025     0.0055     1.0000     0.9999     0.0048
    0.0013     0.0078     1.0000     0.9930     1.0000
    0.0029     0.9999     0.0000     0.0008     0.0002
    0.0000     0.9978     0.0000     0.9991*    0.9051
    0.0000     1.0000     0.0000     1.0000     0.0000
    0.0077     0.9983     0.0000     0.9746     0.9999
    0.0054     0.9972     1.0000     0.0000     0.0038
    0.0006     1.0000     1.0000     0.0002     0.9978
    0.0000     1.0000     1.0000     1.0000     0.0004
    0.0000     0.9840     1.0000     0.9736     1.0000
    1.0000     0.0000     0.0000     0.0000     0.0000
    0.9969     0.0021     0.0000     0.0025     0.9968
    0.9924     0.0060     0.0000     0.9925     0.0070
    0.9985     0.0014     0.0000     0.9997     0.9991
    0.9998     0.0112     0.9960     0.0180     0.0547
    0.9985     0.0010     1.0000     0.0017     0.9992
    0.9977     0.0020     0.9999     0.9940     0.0036
    0.9930     0.0089     1.0000     1.0000     0.9982
    0.9990     0.9999     0.0000     0.0010     0.0014
    1.0000     0.9981     0.0000     0.1275     0.0288*
```

```
Charalpha_12 Simulated Output with noisy input (* = noise)
    0.0000     0.0088     0.0000     0.0074     0.0068
    0.0007     0.0132     0.0445     0.1202     0.8470
    0.0084     0.0075     0.0000     1.0000     0.0016
    0.0144     0.0161     0.0010     0.9894     0.9755
    0.0001     0.0226     0.9920     0.0136     0.3668*
    0.0000     0.0359     1.0000     0.0173     0.9581
    0.0044     0.0087     0.9999     0.9983     0.0098
    0.0030     0.0354     1.0000     0.9763     1.0000
    0.0040     0.9959     0.0000     0.0058     0.0046
    0.0000     0.7596     0.0001     0.4879*    0.9149*
    0.0000     0.9999     0.0000     0.9996     0.0002
    0.0017     0.9898     0.0000     0.9502     0.9978
    0.0109     0.9899     1.0000     0.0001     0.0033
    0.0016     0.9998     1.0000     0.0003     0.9755
    0.0000     0.9992     0.9969     0.9979     0.0016
    0.0000     0.9403     1.0000     0.9195     1.0000
    1.0000     0.0022     0.0000     0.0054     0.0005
    0.9938     0.0313     0.0068     0.0542     0.9920
    0.9672     0.0298     0.0000     0.9454     0.0384
    0.9943     0.0076     0.0004     0.9979     0.9995
    0.9992     0.0187     0.9831     0.0737     0.0667
    0.9880     0.0062     1.0000     0.0127     0.9956
    0.9963     0.0100     0.9999     0.9710     0.0221
    0.9769     0.0331     1.0000     0.9926     0.9956
    0.9911     0.9994     0.0000     0.0088     0.0056
    1.0000     0.9903     0.0000     0.1191     0.2833*
```

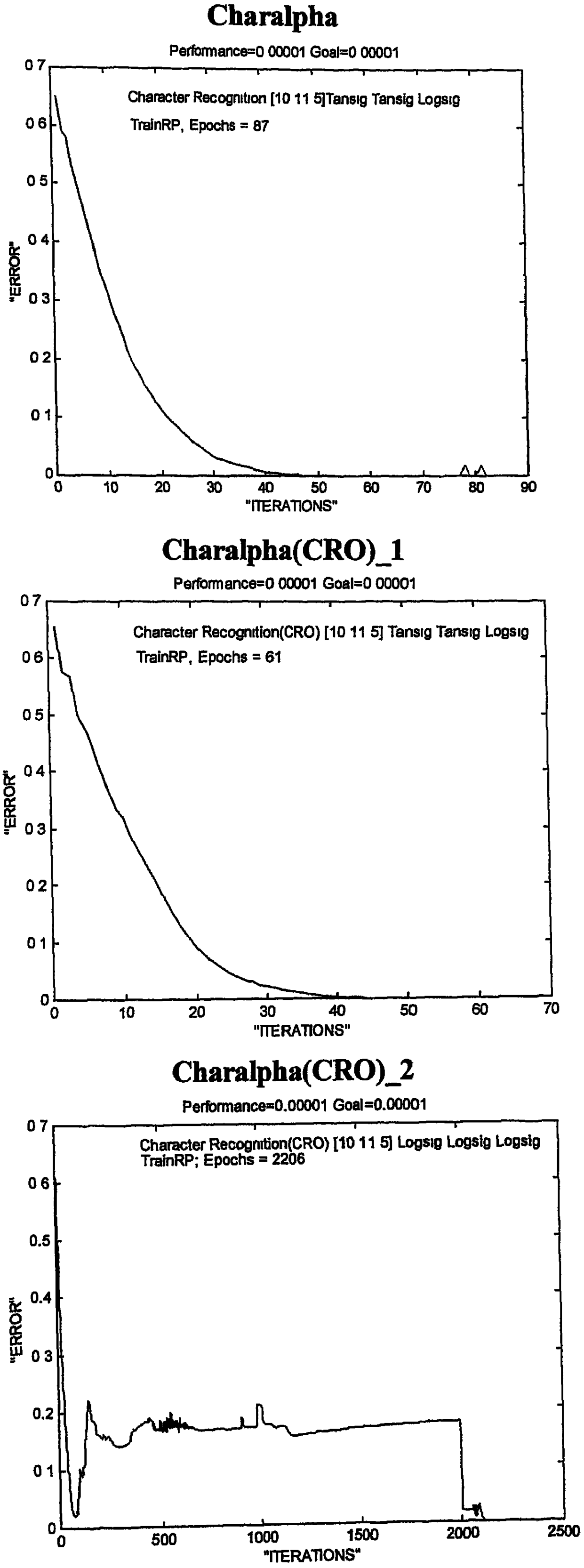
Figure 6.15 : Simulated Output

Charalpha_21 Simulated Output with Noisy Input(* = noise)

```
0.0016    0.0024    0.0009    0.0000    0.0005
0.0056    0.0000    0.0036    0.0039    0.9975
0.0007    0.0006    0.0066    0.9993    0.0013
0.0000    0.0044    0.0016    0.9958    0.9998
0.9965    0.5252    0.0001    0.5658    0.9803*
0.0001    0.0037    0.9961    0.0040    0.9994
0.0000    0.0029    0.9976    0.9986    0.0000
0.0007    0.0006    0.9997    0.9984    0.9977
0.0021    0.9999    0.0000    0.0000    0.0027
0.0170    1.0000    0.0000    0.0000    0.9999*
0.0051    0.9997    0.0043    0.9978    0.0000
0.0018    0.9933    0.0045    0.9923    0.9928
0.0000    0.9989    0.9990    0.0017    0.0000
0.0000    0.9996    0.9996    0.0018    0.9991
0.0000    0.9966    0.9999    0.9939    0.0000
0.0000    0.9959    0.9997    0.9999    0.9995
0.9960    0.0000    0.0024    0.0008    0.0020
0.9989    0.0000    0.0026    0.0025    0.9984
0.9946    0.0008    0.0019    0.9978    0.0009
1.0000    0.0025    0.0026    0.9998    0.9970
0.9979    0.0000    1.0000    0.0050    0.0049
1.0000    0.0000    0.9999    0.0000    1.0000
1.0000    0.0000    1.0000    0.9967    0.0029
1.0000    0.0000    0.9907    0.9946    0.9918
0.9995    1.0000    0.0000    0.0005    0.0000
0.9130    0.9997    0.0000    0.0000    1.0000*
```

Figure 6.16 : Simulated Output(Noisy Data)

JAVA PLOT



Charalpha

Charalpha(CRO)_1

Charalpha(CRO)_2

Figure 6.17 : Character Recognition Error Plots

JAVA SIMULATION

CharalphaTTLRP1: Simulated Output with Noisy Input (* = Noise)

```
0.0000  0.0000  0.0000  0.0002  0.0001
0.0000  0.0036  0.0000  0.0047  0.9999
0.0000  0.0002  0.0000  0.9999  0.0000
0.0000  0.0000  0.0000  0.9999  0.9999*
0.0001  0.0000  0.9999  0.0033  0.0010
0.0004  0.0000  0.9999  0.0008  0.9999
0.0000  0.0003  0.9999  0.9999  0.0005
0.0000  0.0000  0.9999  0.9974  0.9970
0.0002  0.9999  0.0000  0.0039  0.0000*
0.0000  0.9999  0.0000  0.0019  0.9986
0.0000  0.9999  0.0000  0.9999  0.0000
0.0000  0.9999  0.0000  0.9932  0.9996
0.0000  0.9999  0.9999  0.0000  0.0003
0.0000  0.9921  0.9977  0.0000  0.9973
0.0000  0.9977  0.9999  0.9999  0.0000
0.0000  0.9990  0.9999  0.9975  0.9999
0.9999  0.0000  0.0005  0.0010  0.0008
0.9969  0.0000  0.0006  0.0008  0.9999
0.9999  0.0000  0.0019  0.9997  0.0001
0.9988  0.0005  0.0000  0.9999  0.9999
0.9999  0.0017  1.0000  0.0013  0.0012
0.9999  0.0000  0.9983  0.0000  0.9998
0.9999  0.0000  0.9997  0.9993  0.0006
1.0000  0.0008  0.9999  0.9977  0.9967
0.9999  0.9999  0.0000  0.0004  0.0032
0.9999  0.9999  0.0000  0.0006  0.9985*
```

CharalphaVTTLRP1

```
0.0000  0.0000  0.0010  0.0008  0.0000
0.0000  0.0000  0.0021  0.0022  0.9986
0.0000  0.0005  0.0045  0.9974  0.0000
0.0000  0.0002  0.0000  0.9997  0.9999
0.0000  0.0000  0.9980  0.0000  0.0000*
0.0000  0.0000  0.9963  0.0011  0.9994
0.0000  0.0001  0.9998  0.9995  0.0002
0.0000  0.0008  0.9999  0.9991  0.9999
0.0000  0.9998  0.0000  0.0011  0.0000
0.0000  0.9999  0.0000  0.0003  0.9998*
0.0000  0.9999  0.0000  0.9999  0.0001
0.0000  0.9999  0.0019  0.9976  0.9999
0.0000  0.9999  0.9999  0.0000  0.0000
0.0000  0.9999  0.9992  0.0010  0.9995
0.0000  0.9998  0.9926  0.9999  0.0000
0.0000  0.9989  0.9957  0.9999  0.9999
0.9991  0.0001  0.0024  0.0012  0.0000
0.9989  0.0000  0.0067  0.0008  0.9999
0.9991  0.0000  0.0038  0.9972  0.0006
0.9999  0.0000  0.0000  0.9997  0.9986
0.9999  0.0000  0.9999  0.0009  0.0000
0.9999  0.0000  0.9924  0.0000  0.9983
0.9997  0.0000  0.9999  0.9999  0.0003
0.9999  0.0000  0.9999  0.9995  0.9999
0.9999  0.9999  0.0000  0.0000  0.0003
0.9999  0.9999  0.0000  0.4607  0.9658*
```

Figure 6.18: Simulated Output (Noisy Data)

JAVA SIMULATION

CharalphaVLLLRP1: Simulated Output For Noisy Input (* = noise)

```
0.0000  0.0000  0.0000  0.0067  0.0000
0.0000  0.0000  0.0000  0.0016  0.9999
0.0000  0.0000  0.0078  0.9999  0.0000
0.0000  0.0000  0.0000  0.9999  0.9999
0.0000  0.0000  0.9999  0.0030  0.0004*
0.0000  0.0000  0.9999  0.0008  0.9999
0.0000  0.0000  1.0000  0.9934  0.0000
0.0000  0.0000  0.9999  0.9999  0.9956
0.0000  1.0000  0.0000  0.0002  0.0000
0.0000  1.0000  0.0000  0.0000  0.9999*
0.0000  1.0000  0.0000  0.9999  0.0000
0.0000  1.0000  0.0005  0.9996  0.9999
0.0010  1.0000  0.9999  0.0006  0.0000
0.0000  1.0000  0.9999  0.0004  0.9999
0.0000  0.9999  0.9999  0.9999  0.0000
0.0000  1.0000  0.9999  0.9985  0.9999
0.9999  0.0000  0.0058  0.0002  0.0000
1.0000  0.0000  0.0000  0.0020  0.9999
0.9999  0.0000  0.0000  0.9928  0.0000
1.0000  0.0000  0.0000  0.9999  0.9945
0.9998  0.0000  1.0000  0.0108  0.0000
0.9999  0.0000  1.0000  0.0029  0.9999
1.0000  0.0000  0.9999  0.9999  0.0024
1.0000  0.0000  0.9998  0.9995  0.9987
1.0000  1.0000  0.0000  0.0007  0.0001
0.9999  1.0000  0.0001  0.0000  0.9999*
```

Figure 6.19 : Simulated Output (Noisy Data)

6.5 The SinxSiny Problem

This is a problem of function approximation. Sin(x)Sin(y) is a complicated graph and capturing it using Artificial neural network requires a very systematic search for the right network architecture. Care has to be taken with regards error tolerance, a very stringent error requirement may lead to over-fitting, and if the error requirement is kept very loose then the network may not capture the curve sufficiently to carry out interpolation. In this work a total of 625 points have been generated (x and y varying from 0 to 2 pi), out of these 100 points were used for training and the trained network was used to simulate the remaining points. The results and plots are shown in figures 6.20 to 6.33 (MATLAB) and figures 6.34 to 6.49 (JAVA). The architectures found to give proper capturing of the curve are listed below:

MATLAB RESULTS

Table 6.10: SinxSiny Problem

SinxSiny Problem: ANN Architectures used; Error Goal = 0.0001				
Network	Architecture	Activation Fn.	Algorithm	Iterations
SinxSiny_n	[6 5 6 1]	Tansig Tansig Tansig Logsig	TrainLM	623
SinxIny_n1	[15 15 15 1]	Tansig Tansig Tansig Logsig	TrainSCG	635
SinxSiny_n2	[25 25 1]	Tansig Tansig logsig	TrainSCG	1100
SinxSiny_n3	[20 25 30 1]	Tansig Tansig Tansig Logsig	TrainRP	425
SinxSiny_n4	[35 35 1]	Tansig Tansig logsig	TrainRP	500
SinxSiny_n5	[15 15 15 1]	Tansig Tansig Tansig Logsig	TrainCGB	1200
SinxSiny_n6	[20 25 30 1]	Tansig Tansig Tansig Logsig	TrainCGP	550

TrainSCG = Scaled Conjugate Gradient Back-propagation

TrainCGB = Conjugate Gradient Back-propagation with Powell-Beale restarts

TrainCGP = Conjugate Gradient Back-propagation with Polak-Ribiere updates

JAVA RESULTS

Table 6.11: SinxSiny Problem

SinxSiny Problem: ANN Architectures used; Error Goal = 0.0001				
Network	Architecture	Activation Fn.	Algorithm	Iterations
SinxIny_n1	[15 15 15 1]	Tansig Tansig Tansig Tansig	TrainSCG	2767
SinxSiny_n11	[15 15 15 1]	Tansig Tansig Tansig Logsig	TrainSCG	2338
SinxSiny_n2	[25 25 1]	Tansig Tansig Tansig	TrainSCG	3000
SinxSiny_n21	[25 25 1]	Tansig Tansig logsig	TrainSCG	7760
SinxSiny_n3	[20 25 30 1]	Tansig Tansig Tansig Logsig	TrrainRP	801
SinxSiny_n4	[35 35 1]	Tansig Tansig Tansig	TrainRP	5000

Conclusion: From the achieved results the following are concluded:-

(a) Sinxsiny requires very rigorous error goal.

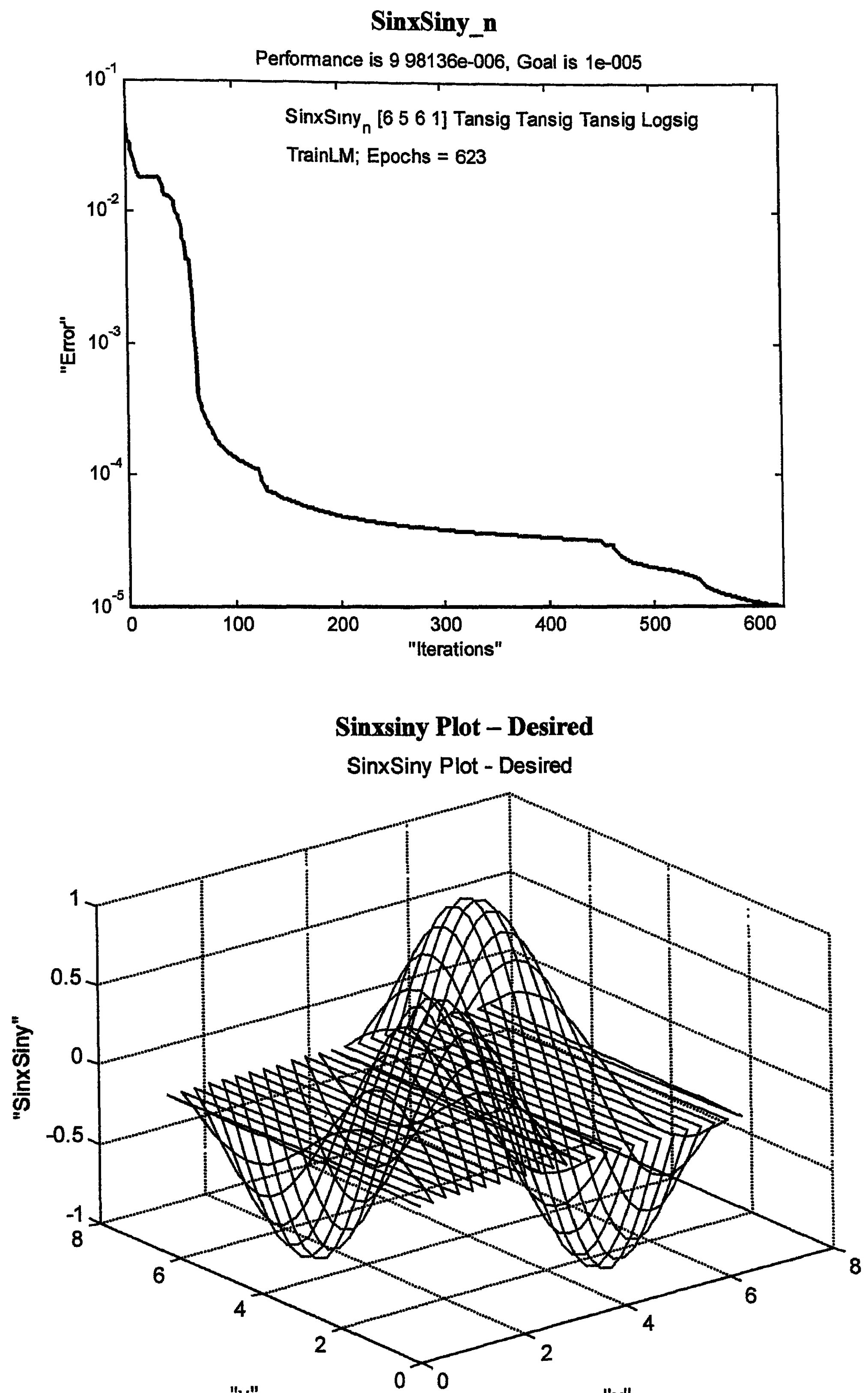
(b) The number of neurons required is very large.

(c) Only Levenberg-Marquardt second order algorithm requires less number of neurons.

(d) Both Trainrp and trainscg take longer time and iterations to converge, in the case of JAVA.

(e) The curve fitting was better in the case of JAVA.

(f) Input data set requires the product term (x*y) also to enable the network to capture the curve properly.

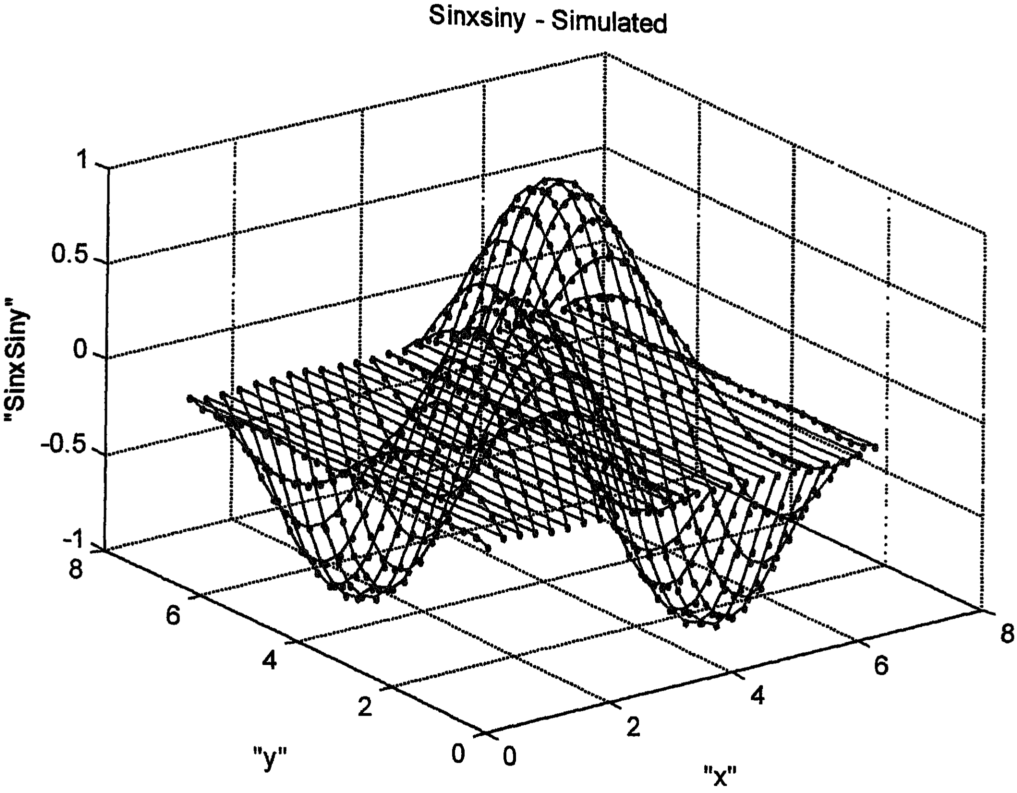


SinxSiny_n

Sinxsiny Plot – Desired

Figure 6.20: Sin(x)Sin(y) Plots

SinxSiny_n



SinxSiny Plot – Combined

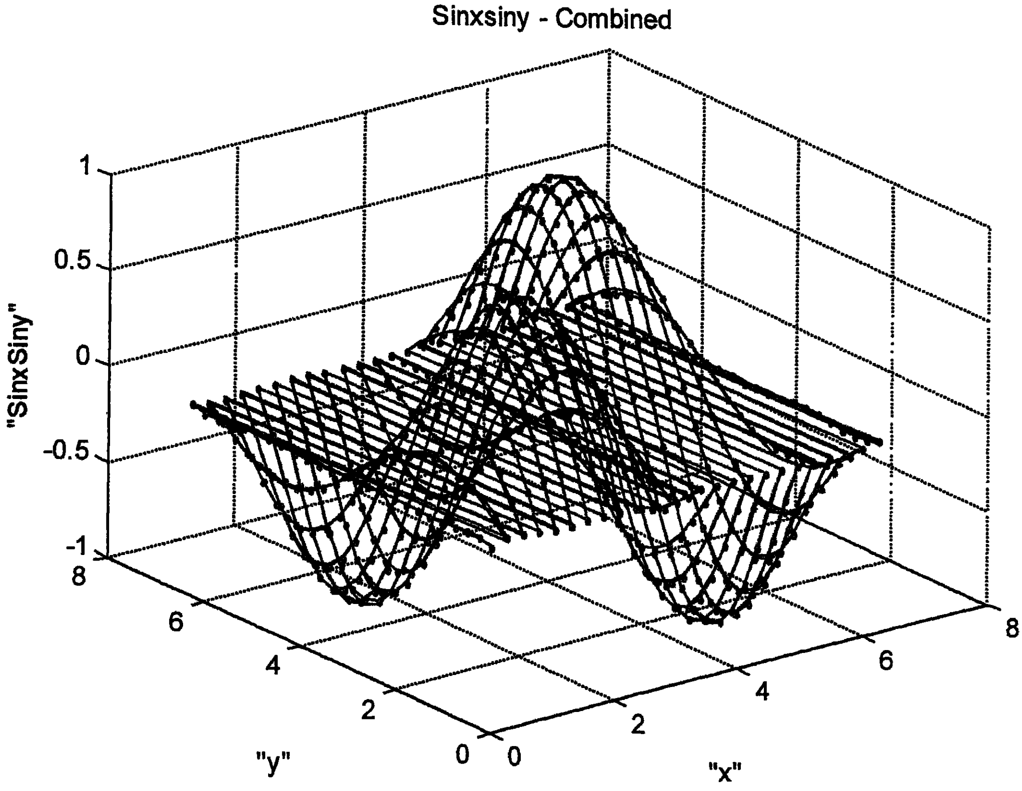
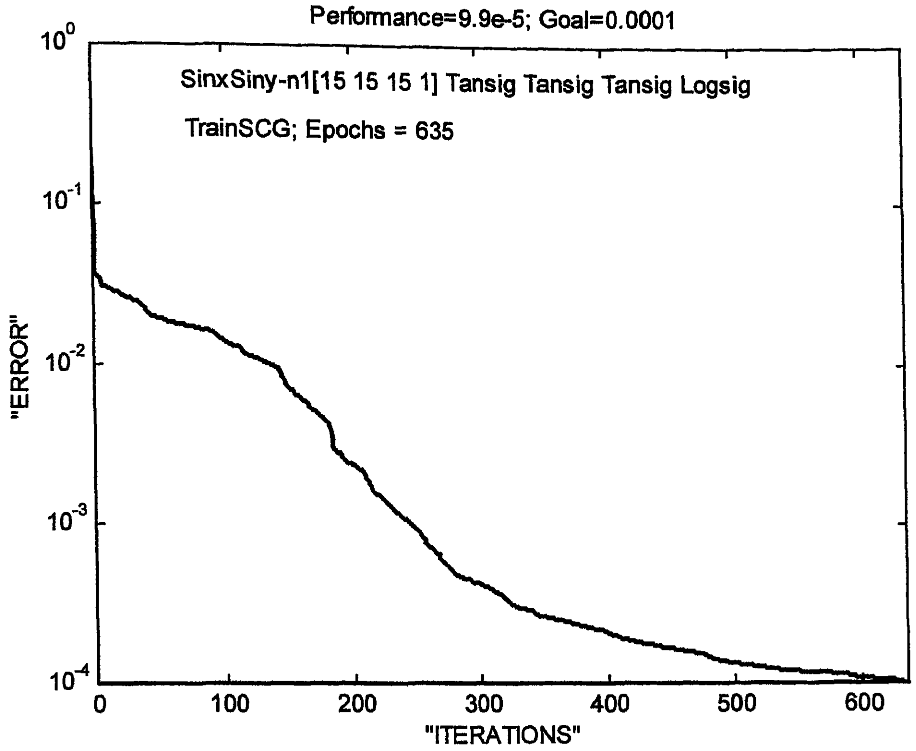


Figure 6.21 : SinxSiny Plots

Sinxsiny_n1



SinxSiny Plot - Desired

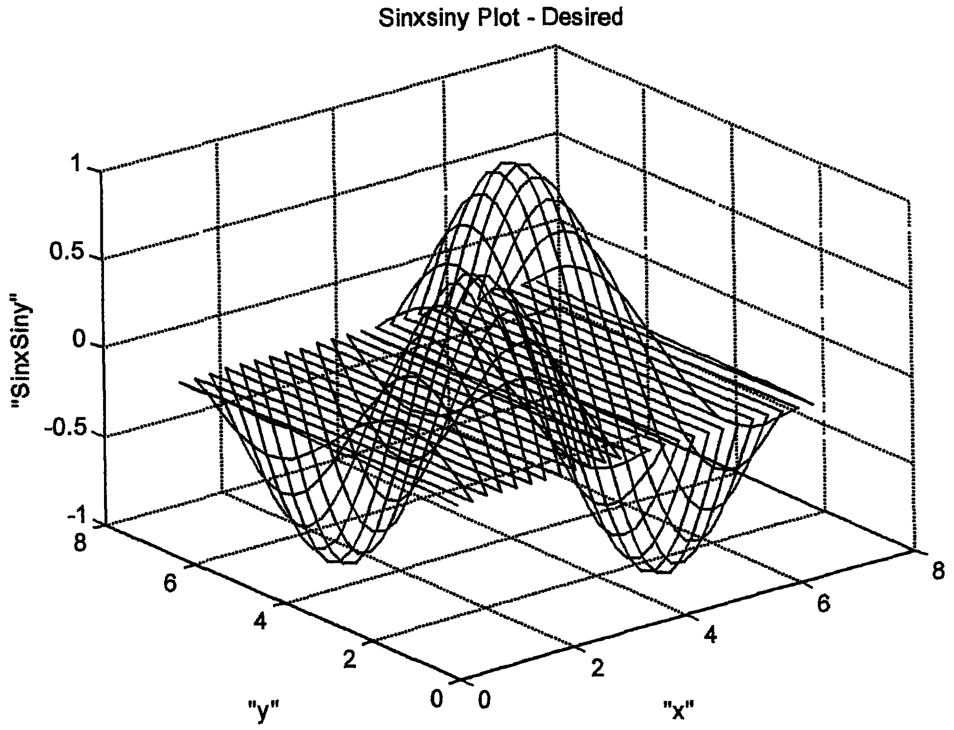
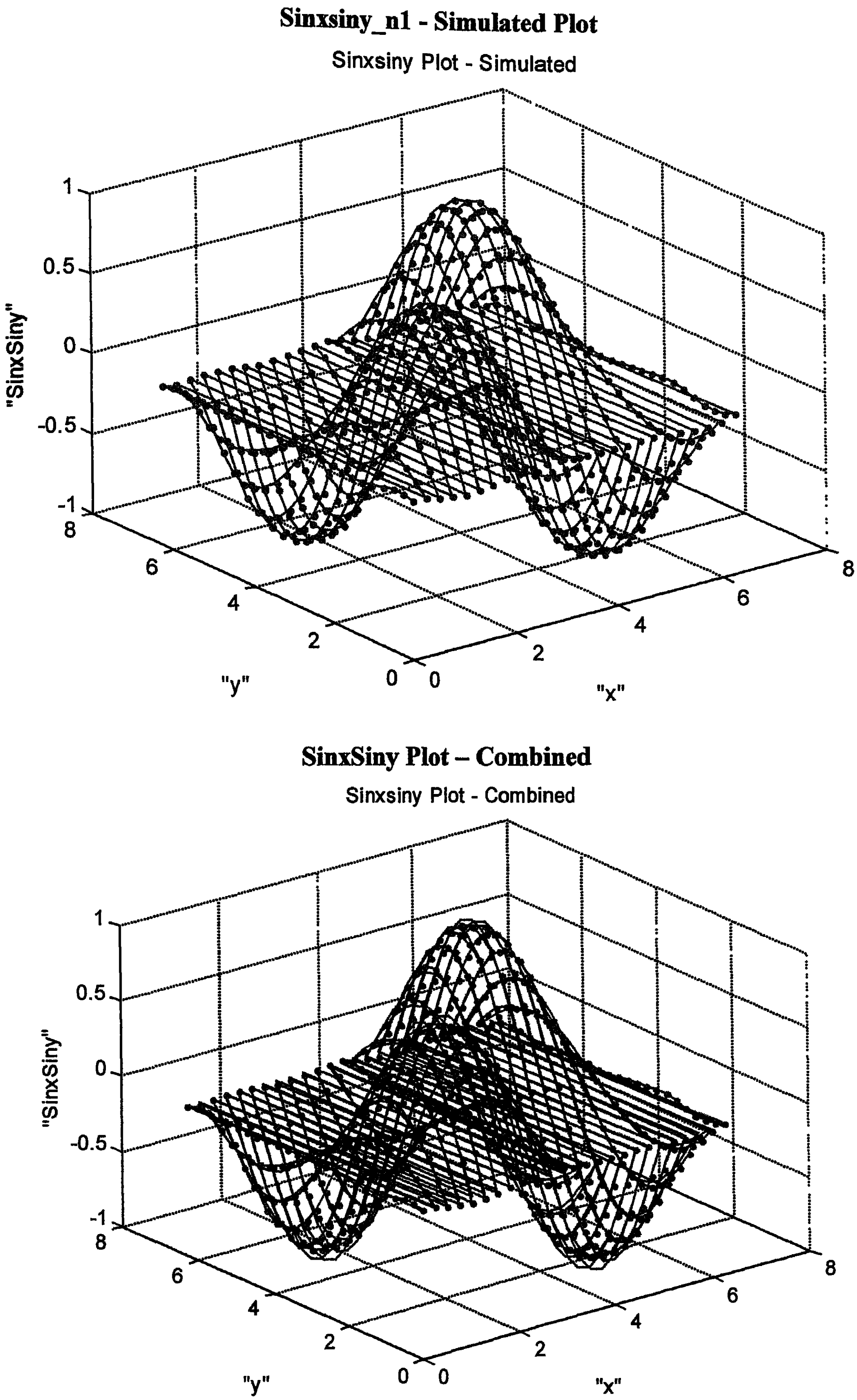


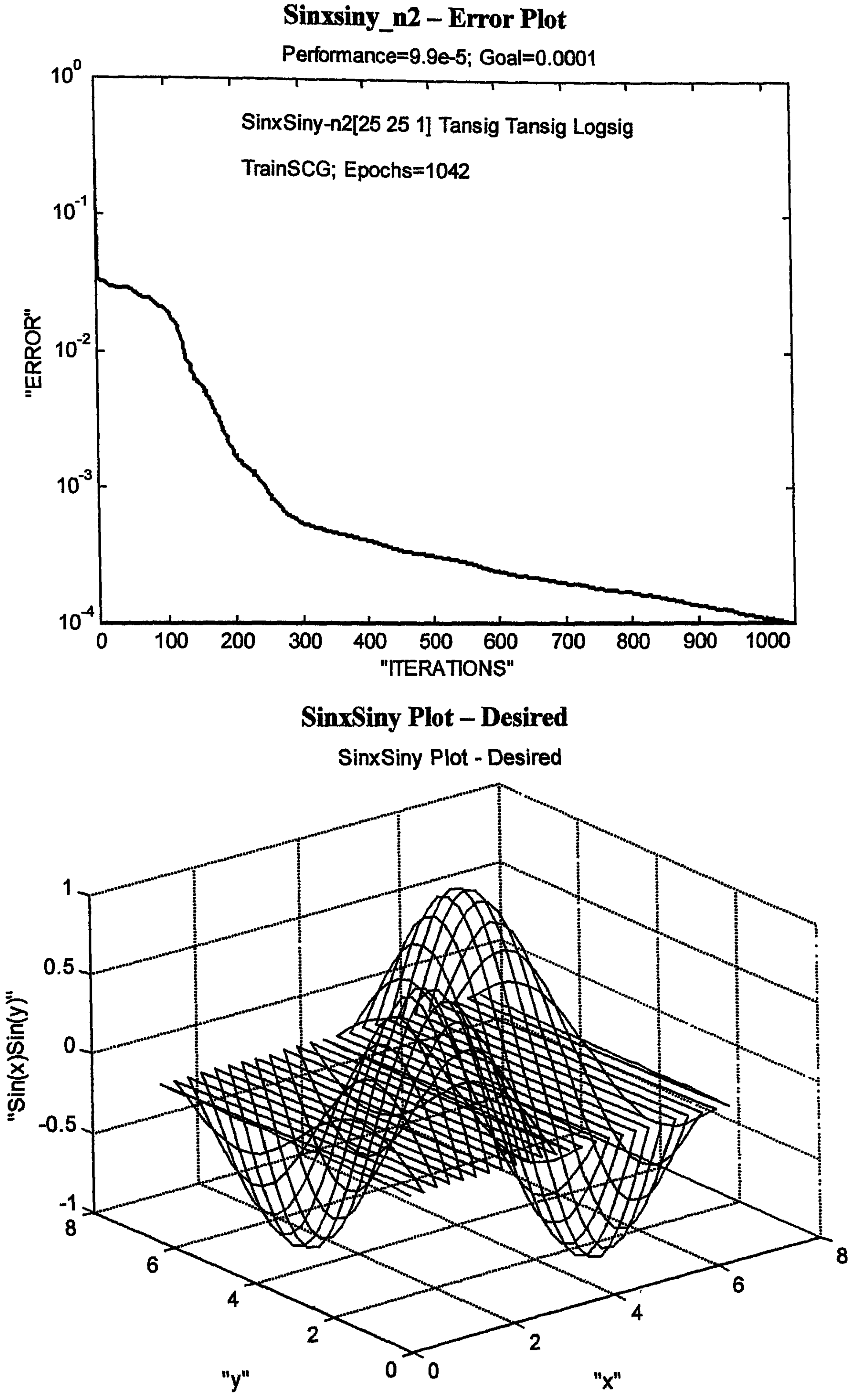
Figure 6.22: Sinxsiny Plots



Sinxsiny_n1 - Simulated Plot

SinxSiny Plot – Combined

Figure 6.23 : Sin(x)Sin(y) Plots



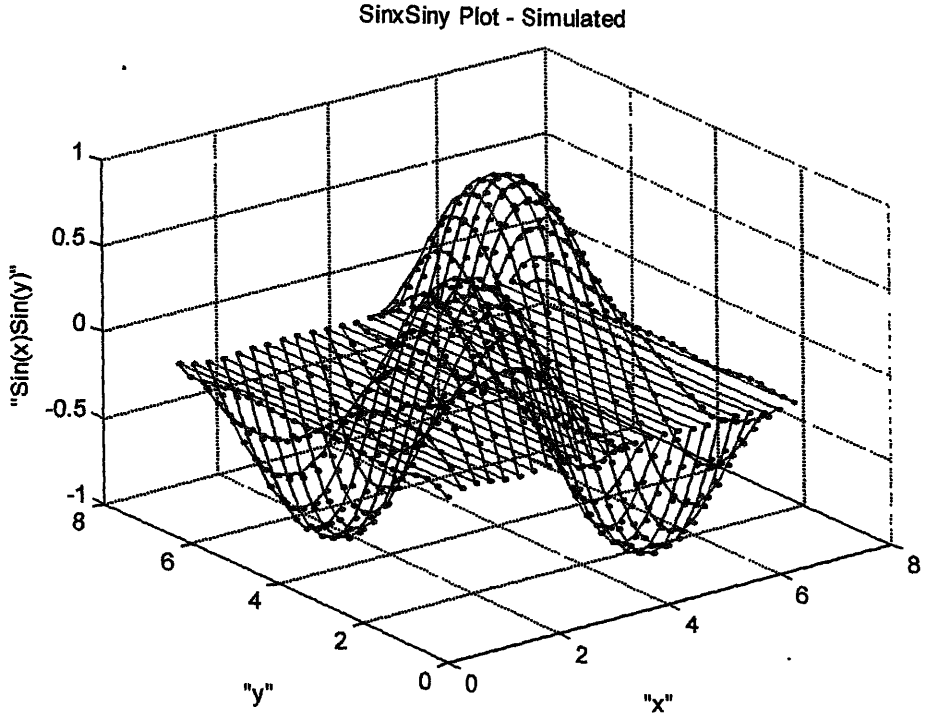
Sinxsiny_n2 – Error Plot

SinxSiny Plot – Desired

Figure 6.24 : SinxSiny Plot

Sinxsiny_n2

SinxSiny Plot – Simulated



SinxSiny Plot – Combined

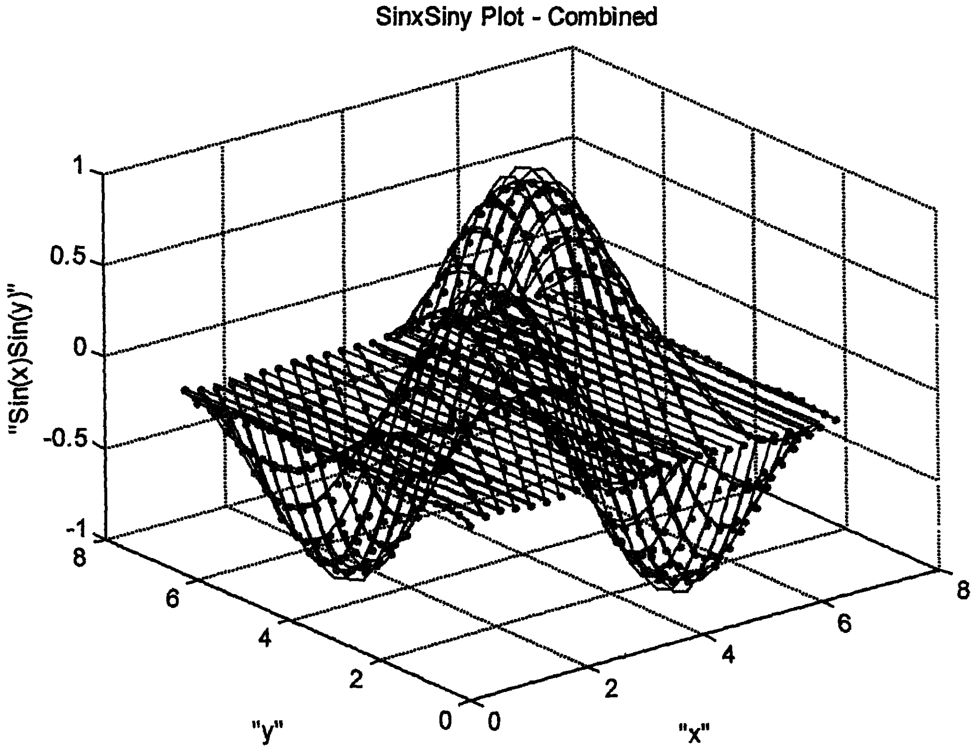
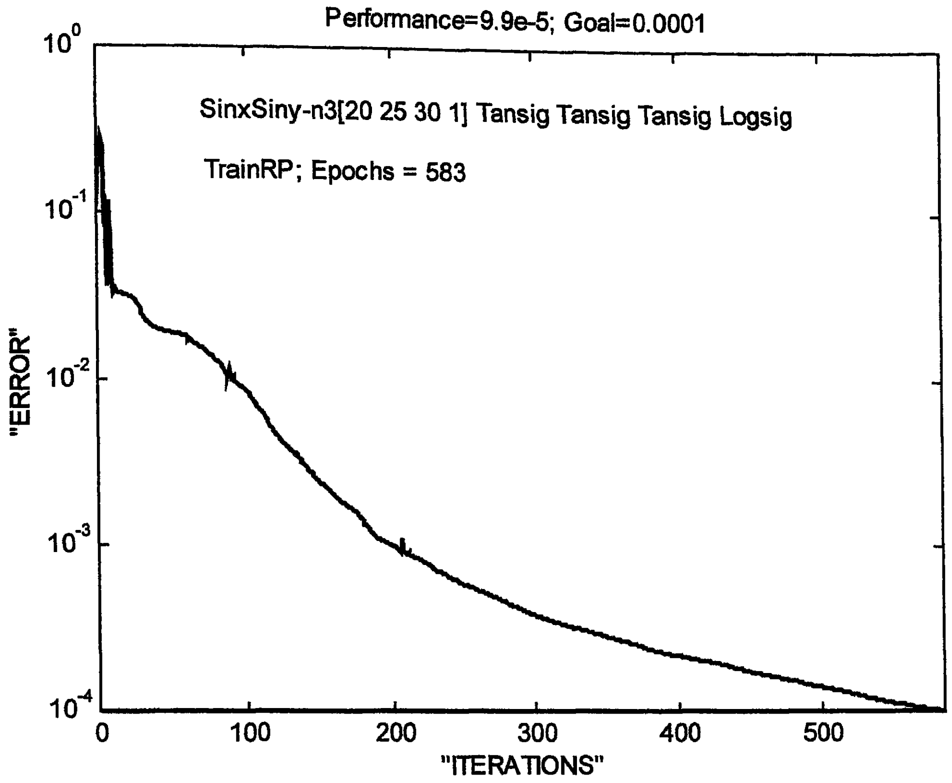


Figure 6.25: Sin(x)Sin(y) Plots

Sinxsiny_n3

SinxSiny - Error Plot



SinxSiny Plot – Desired

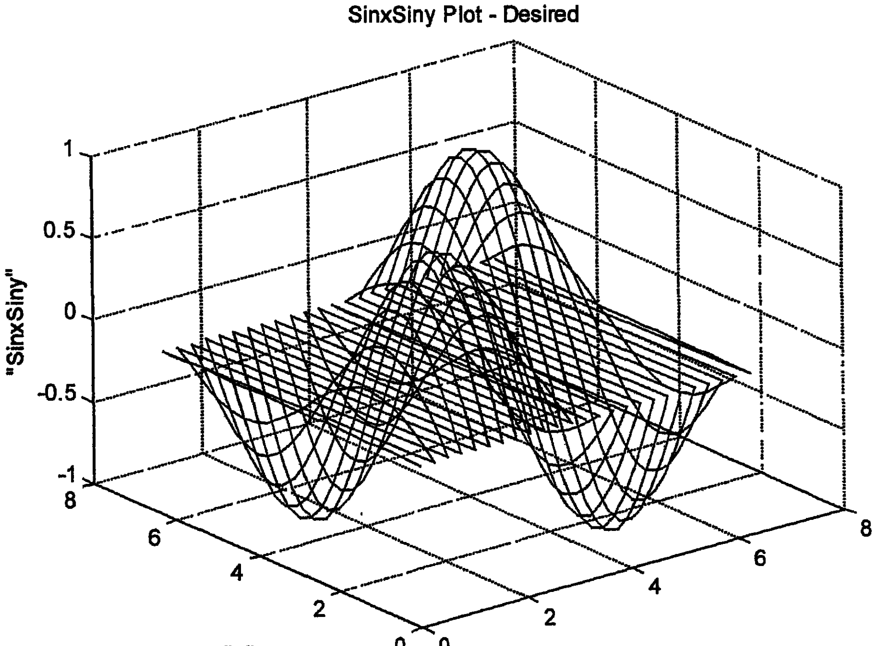
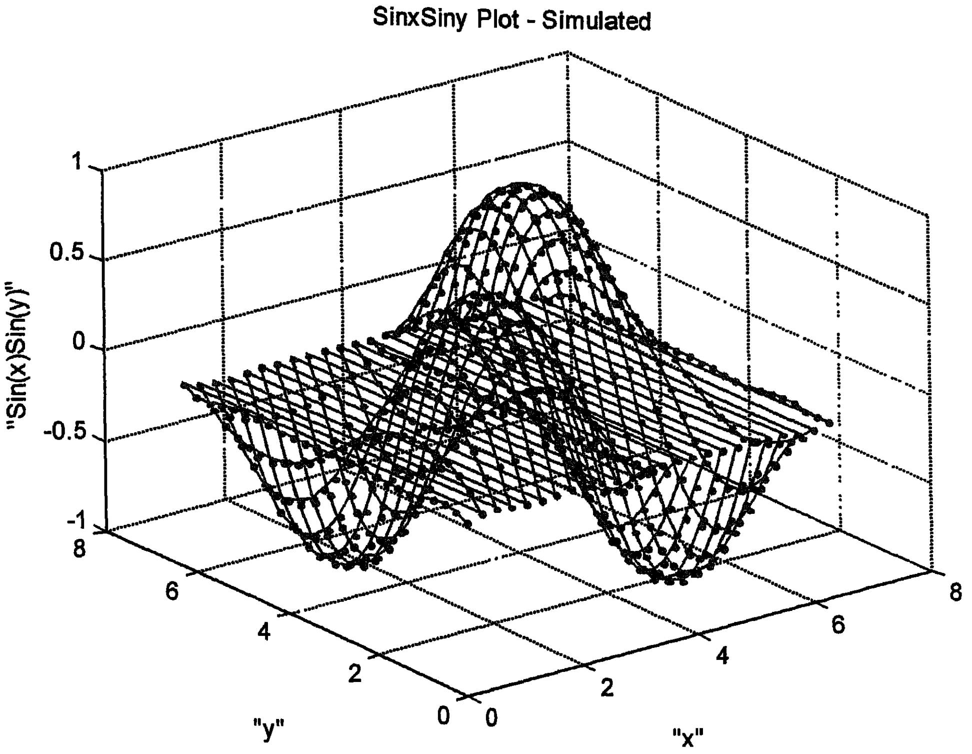


Figure 6.26: Sin(x)Sin(y) Plots

Sinxsiny_n3

SinxSiny Plot – Simulated



SinxSiny Plot – Combined

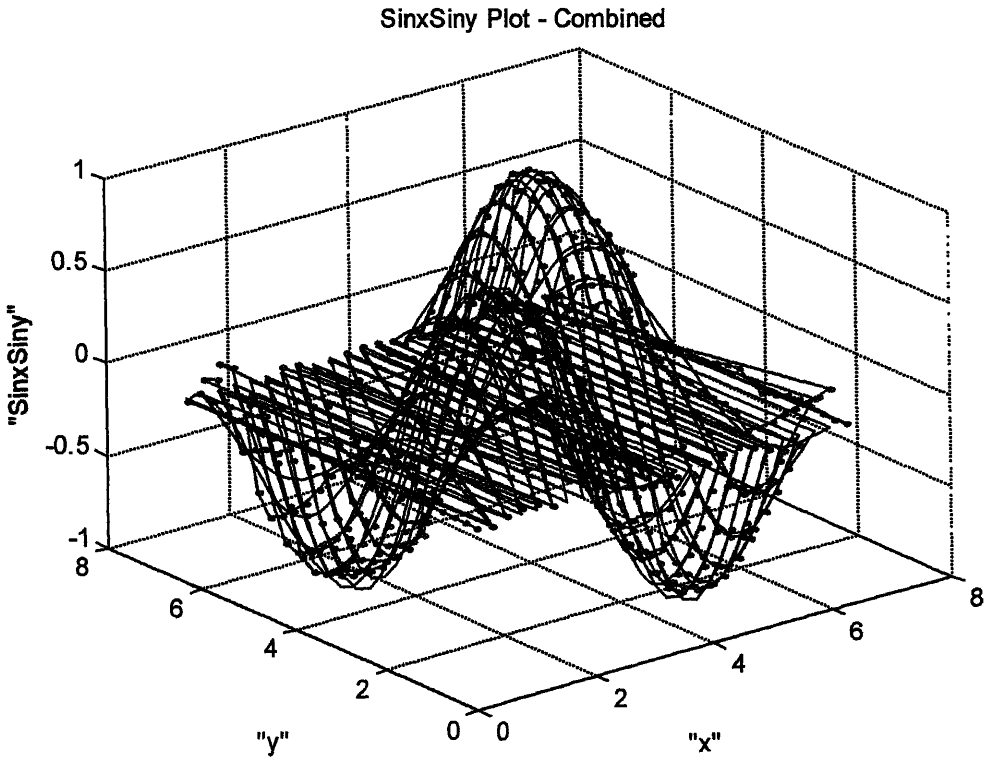
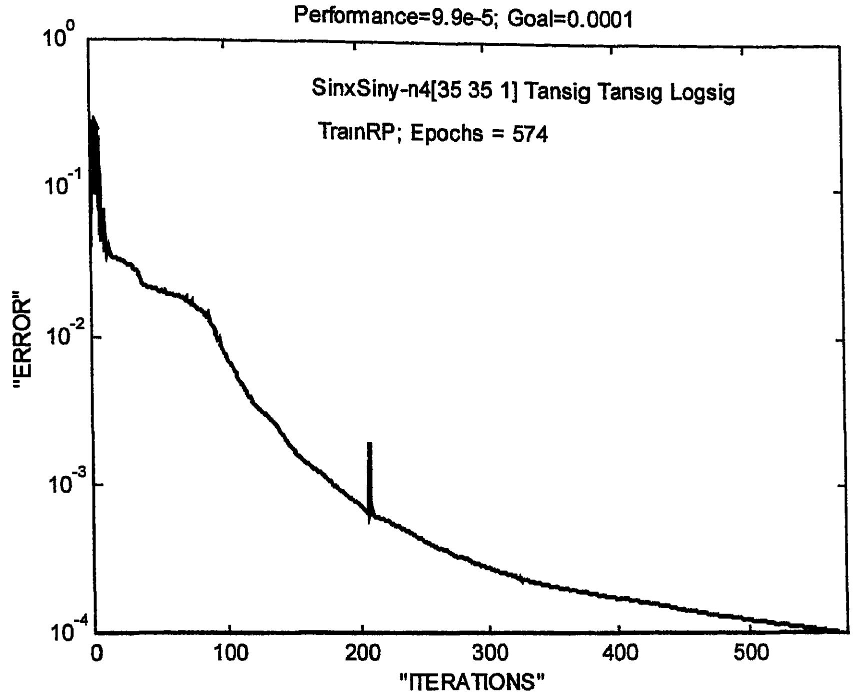


Figure 6.27: Sin(x)Sin(y) Plot

Sinxsiny_n4

SinxSiny - Error Plot



SinxSiny Plot – Desired

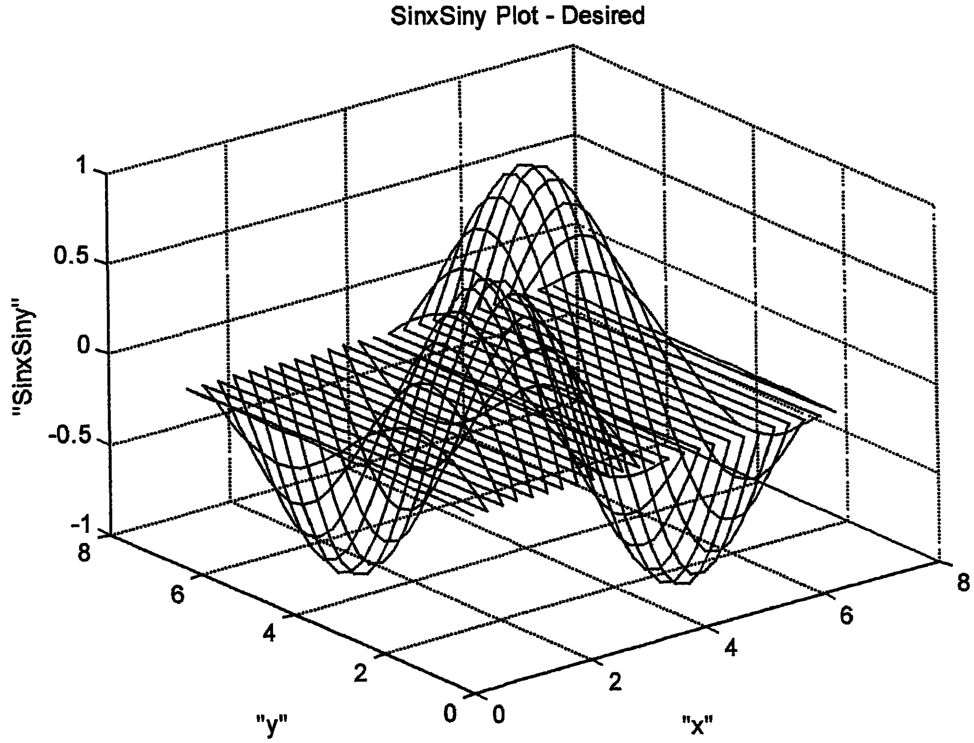
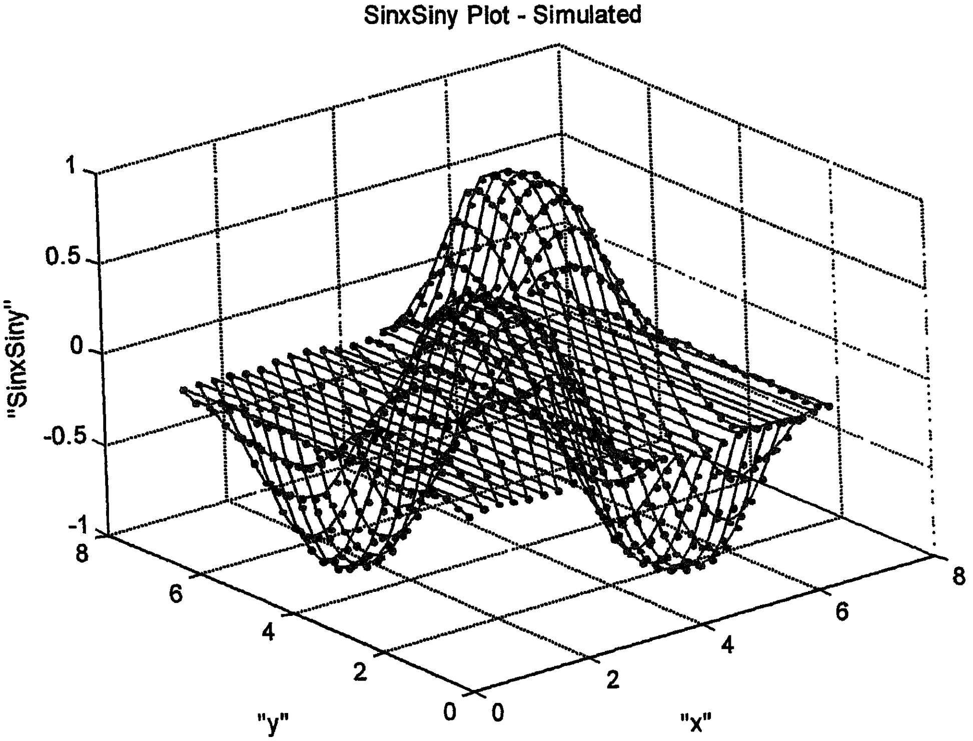


Figure 6.28 : Sin(x)Sin(y) Plots

Sinxsiny_n4

SinxSiny Plot – Simulated



SinxSiny Plot – Combined

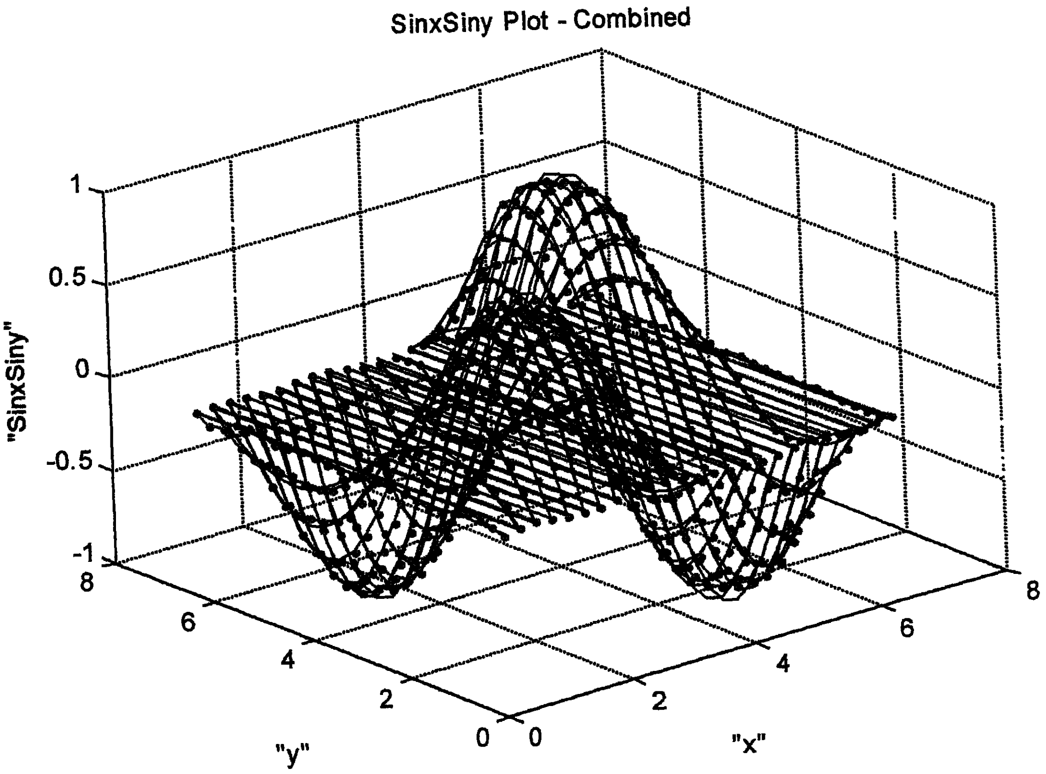
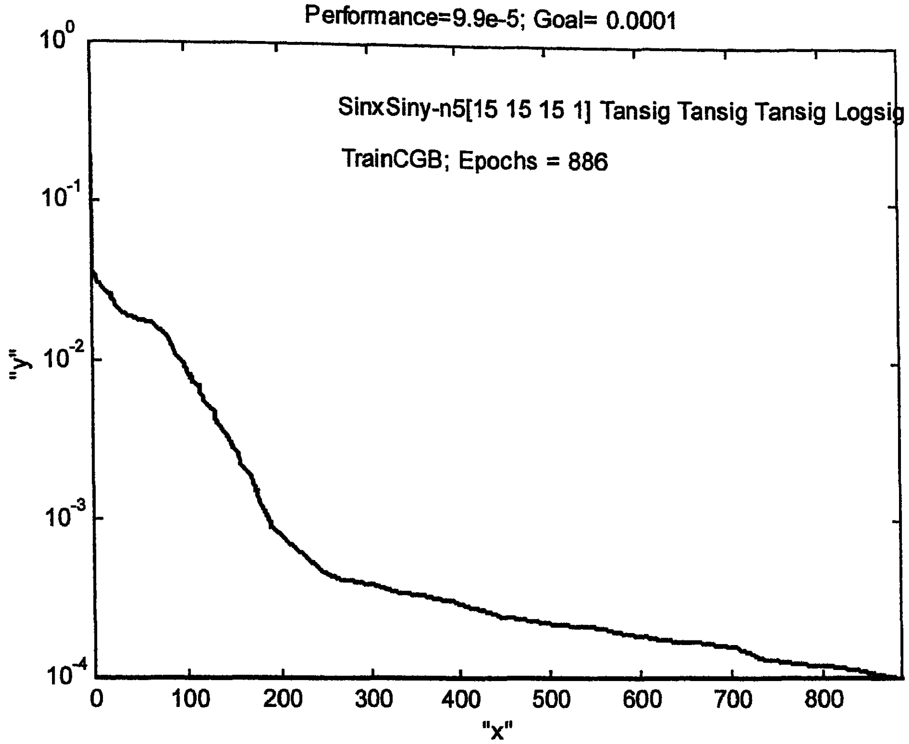


Figure 6.29: Sin(x)Sin(y) Plots

Sinxsiny_n5

SinxSiny Error Plot



SinxSiny Plot – Desired

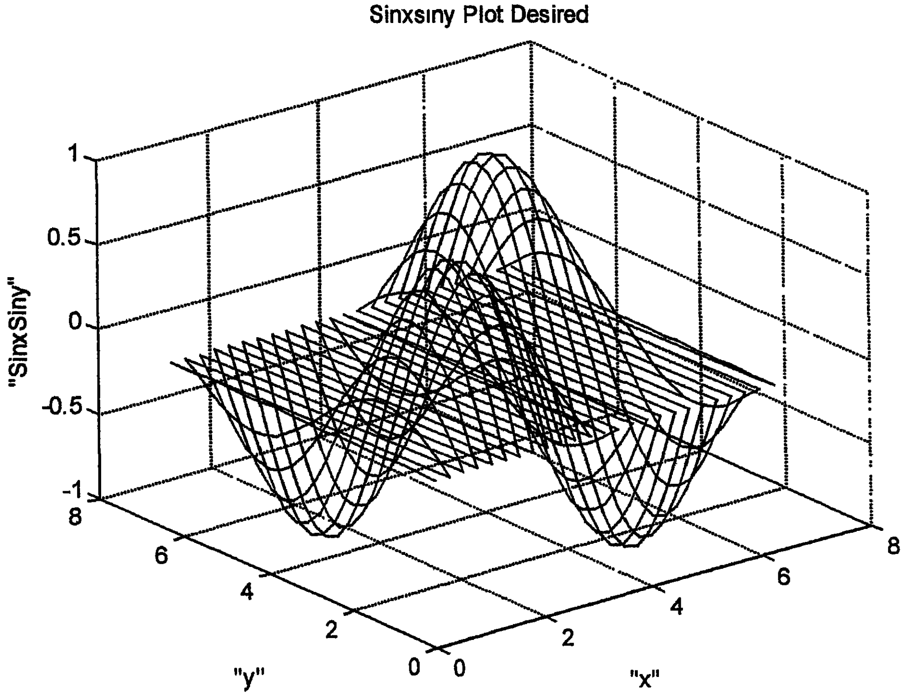
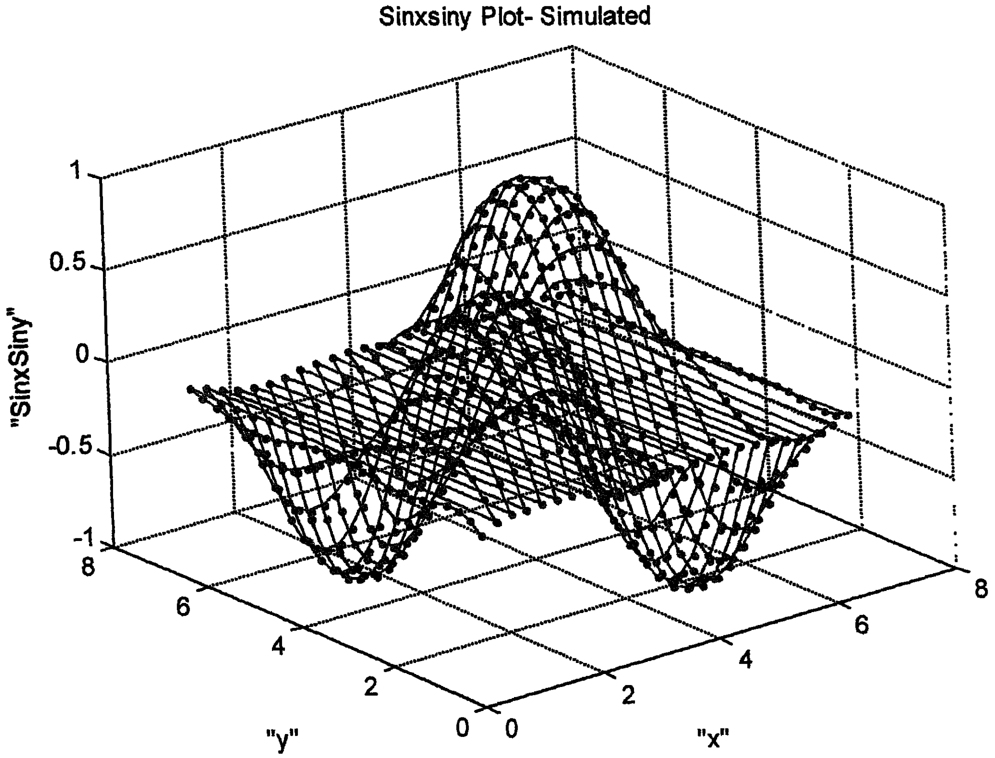


Figure 6.30 : Sin(x)Sin(y) Plots

Sinxsiny_n5

SinxSiny Plot – Simulated



SinxSiny Plot – Combined

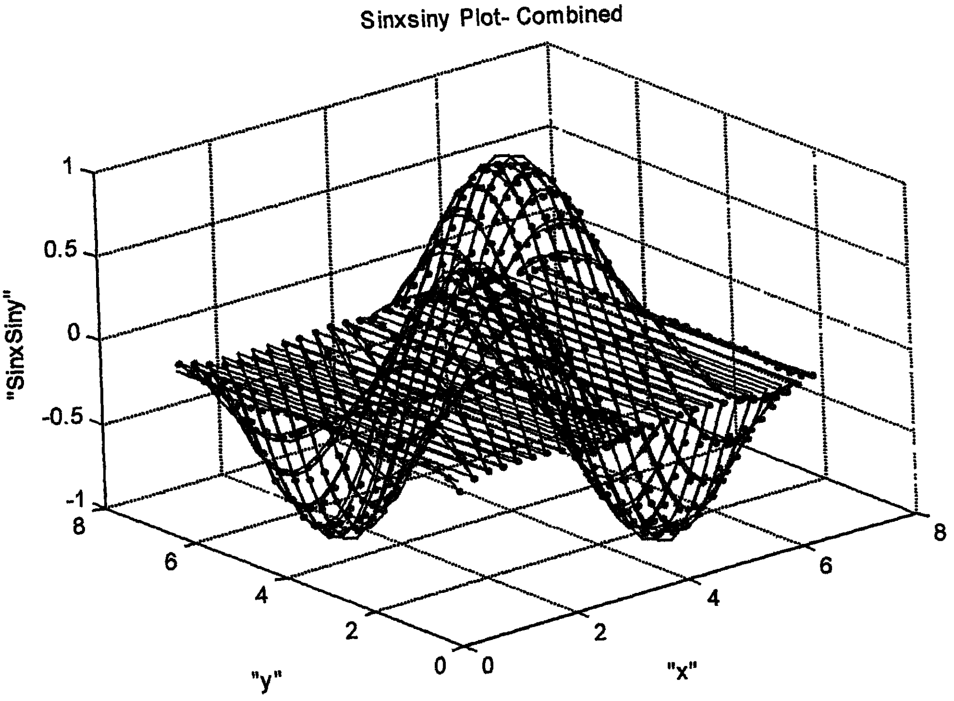
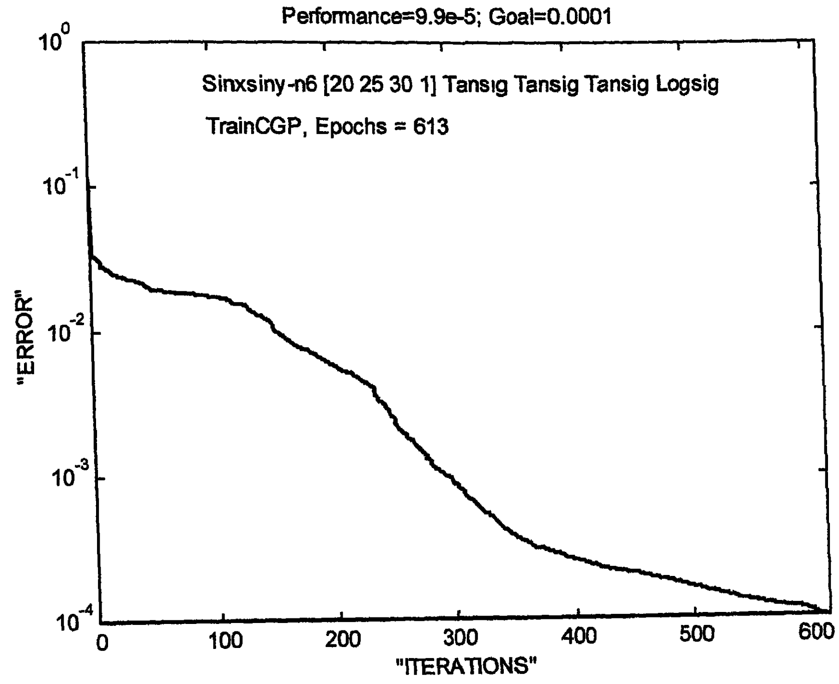


Figure 6.31 : Sin(x)Sin(x) Plots

Sinxsiny_n6

SinxSiny Error Plot



SinxSiny Plot – Desired

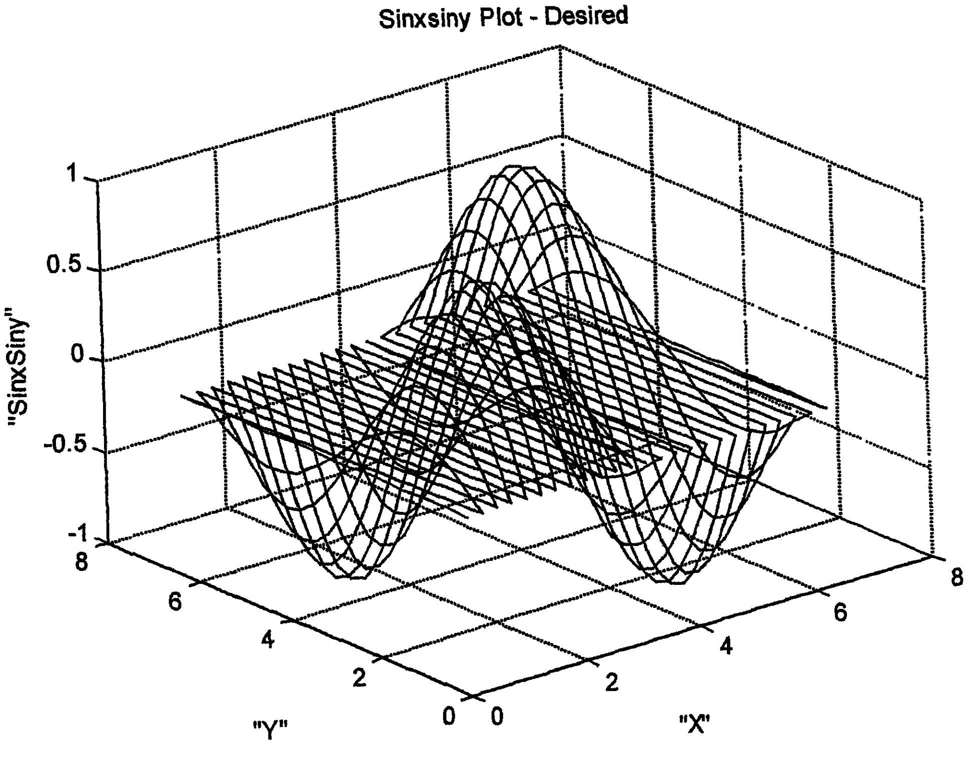
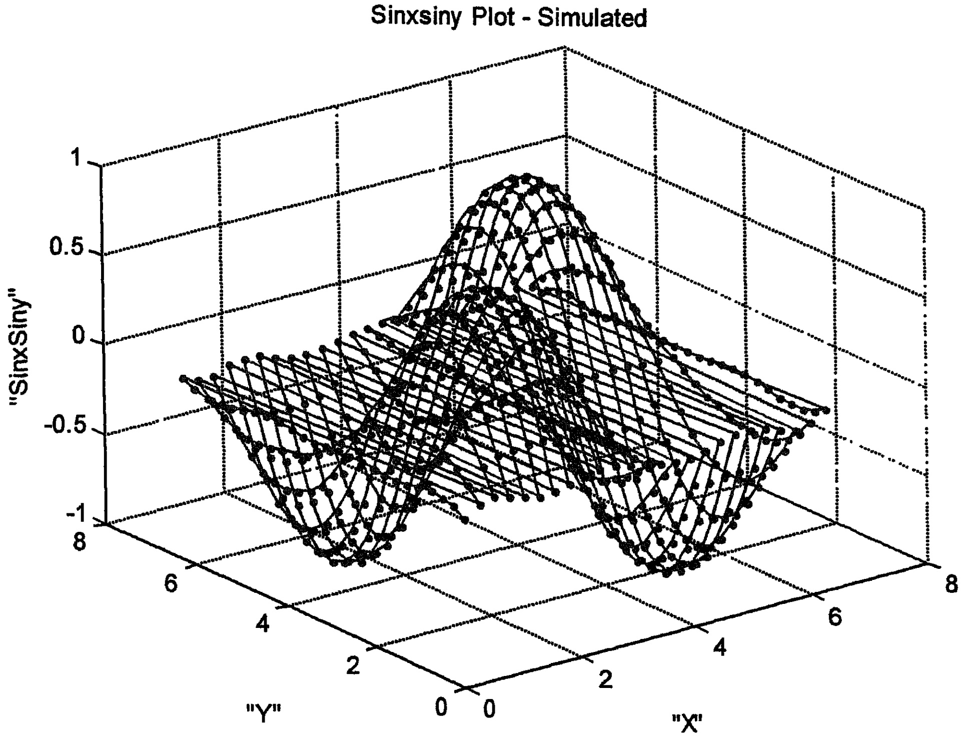


Figure 6.31: Sin(x)Sin(y) Plots

Sinxsiny_n6

SinxSiny Plot – Simulated



SinxSiny Plot – Combined

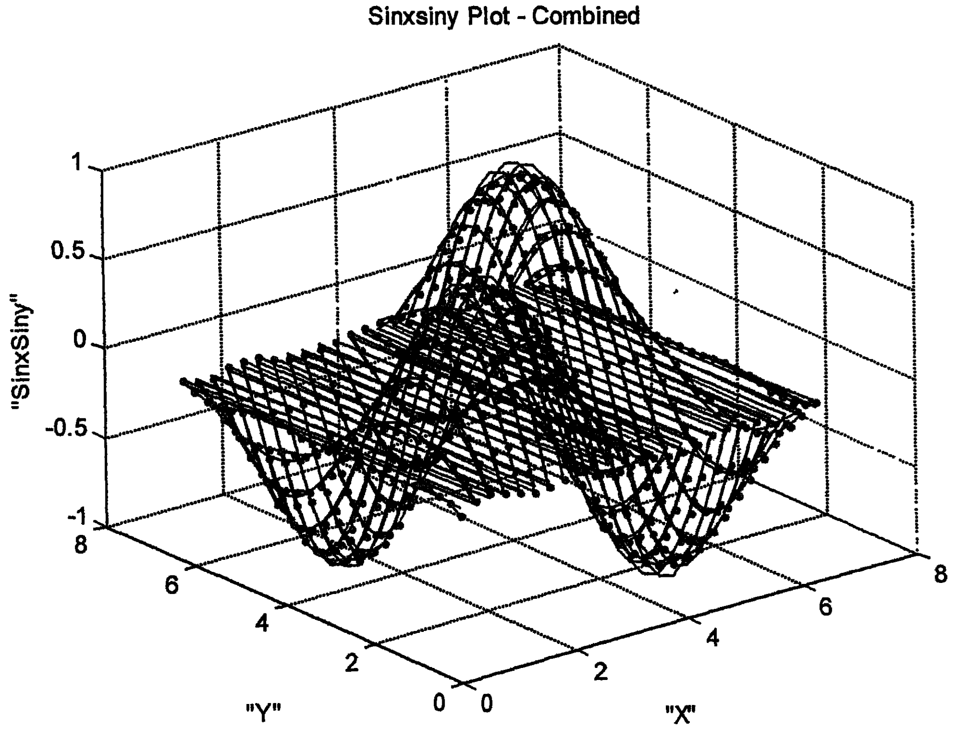


Figure 6.33 : Sin(x)Sin(y) Plots

Sin(x)Sin(y)-Error Plots

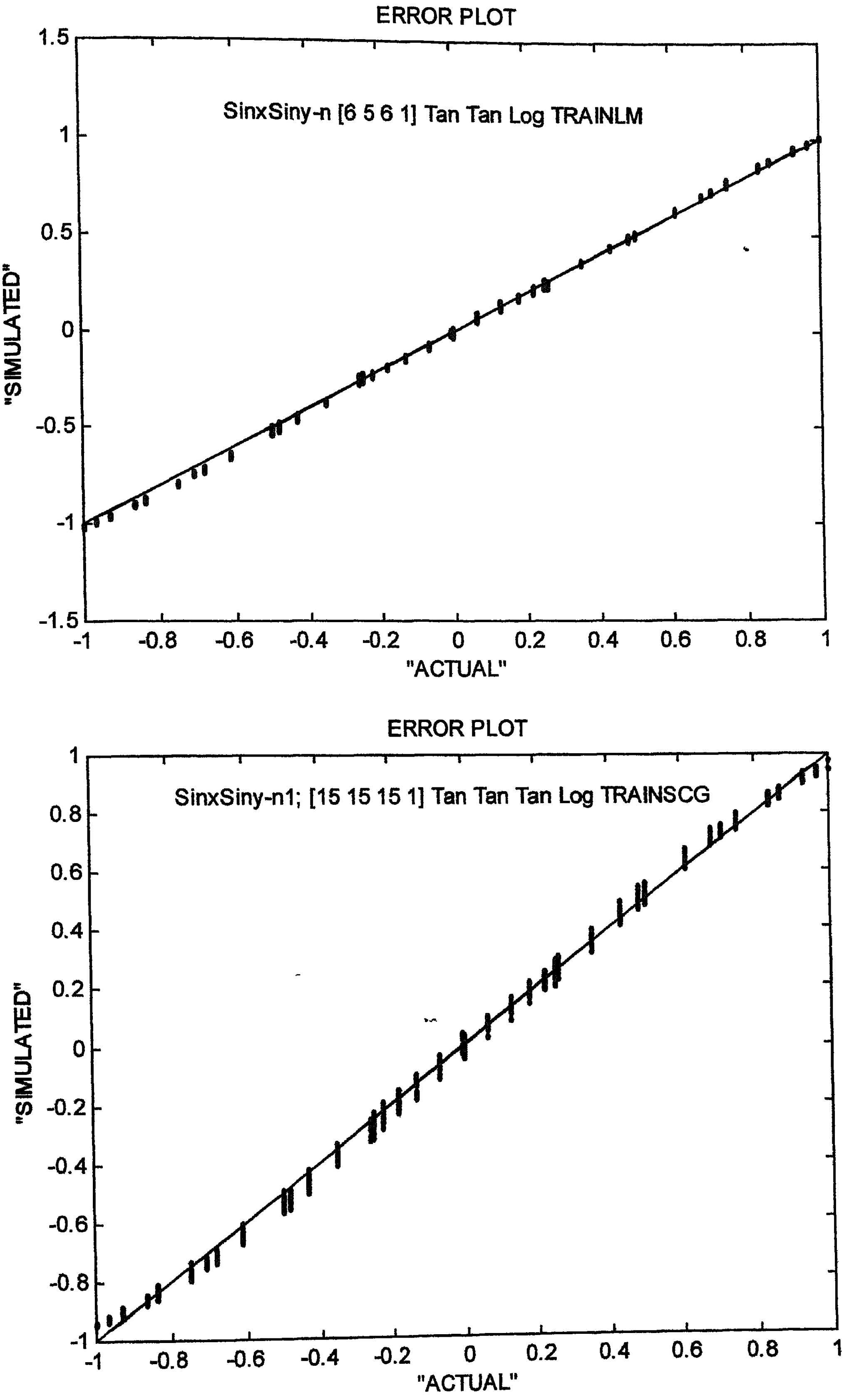
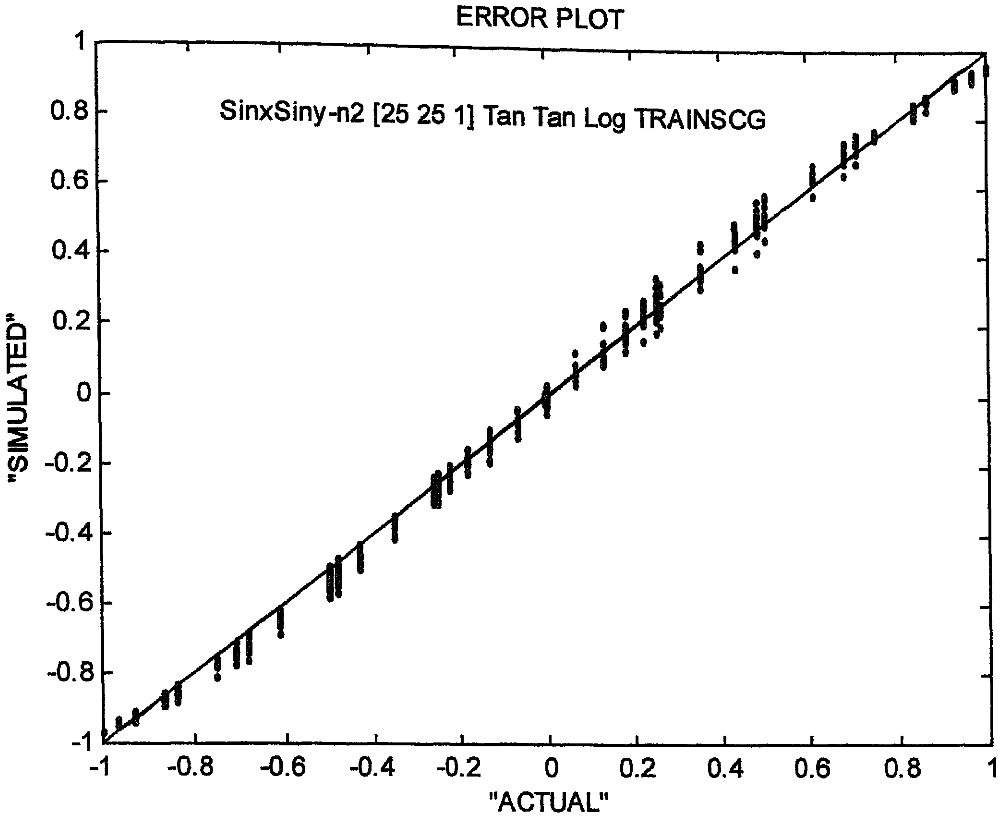


Figure 6.33b; Sin(x)Sin(y) Error Plots(MATLAB)

Sin(x)Sin(y)-Error Plots



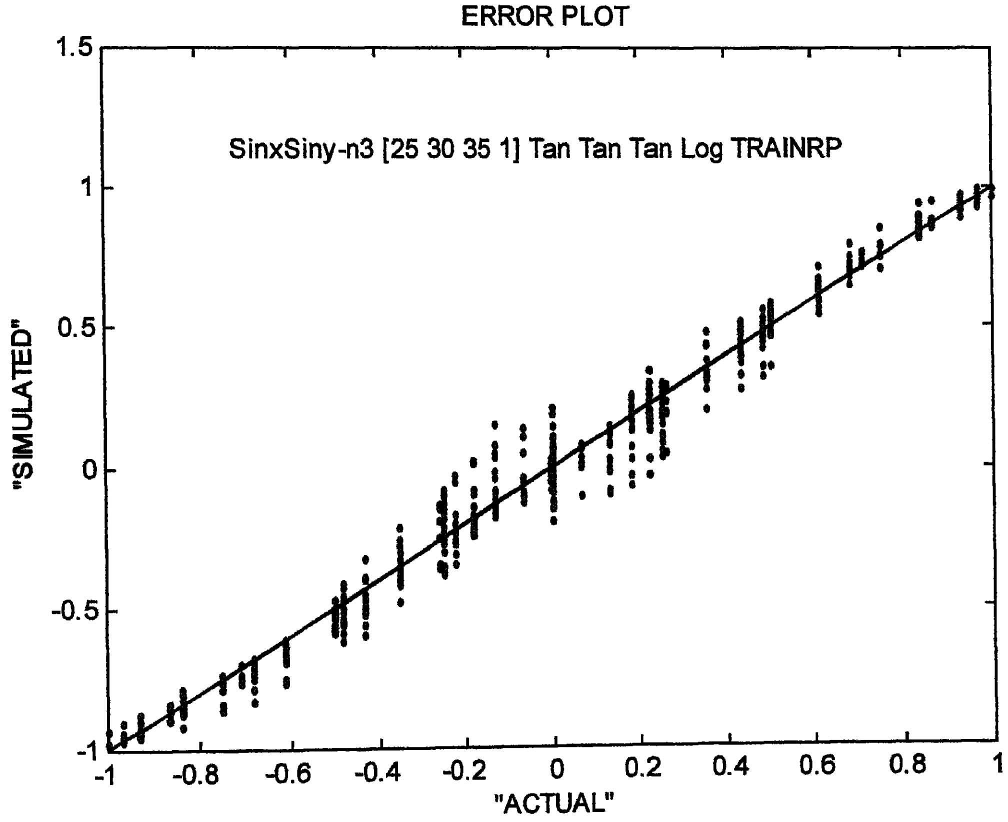
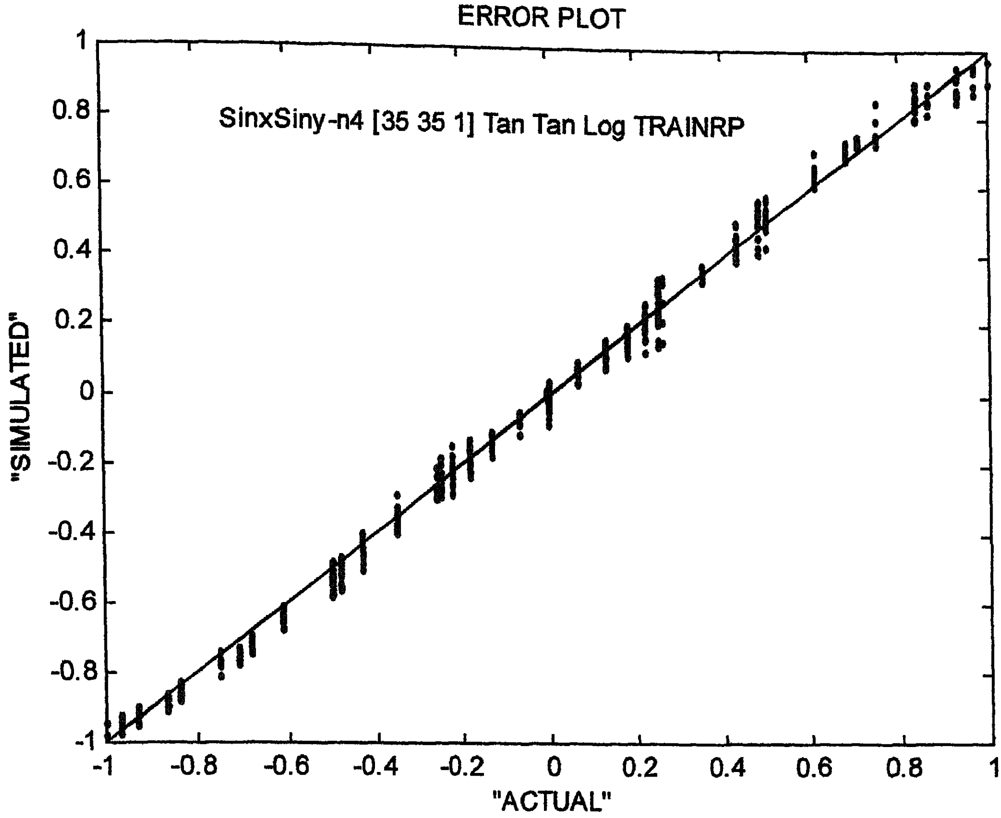


Figure 6.33c Sin(x)Sin(y) Error Plots(MATLAB)

Sin(x)Sin(y)-Error Plots



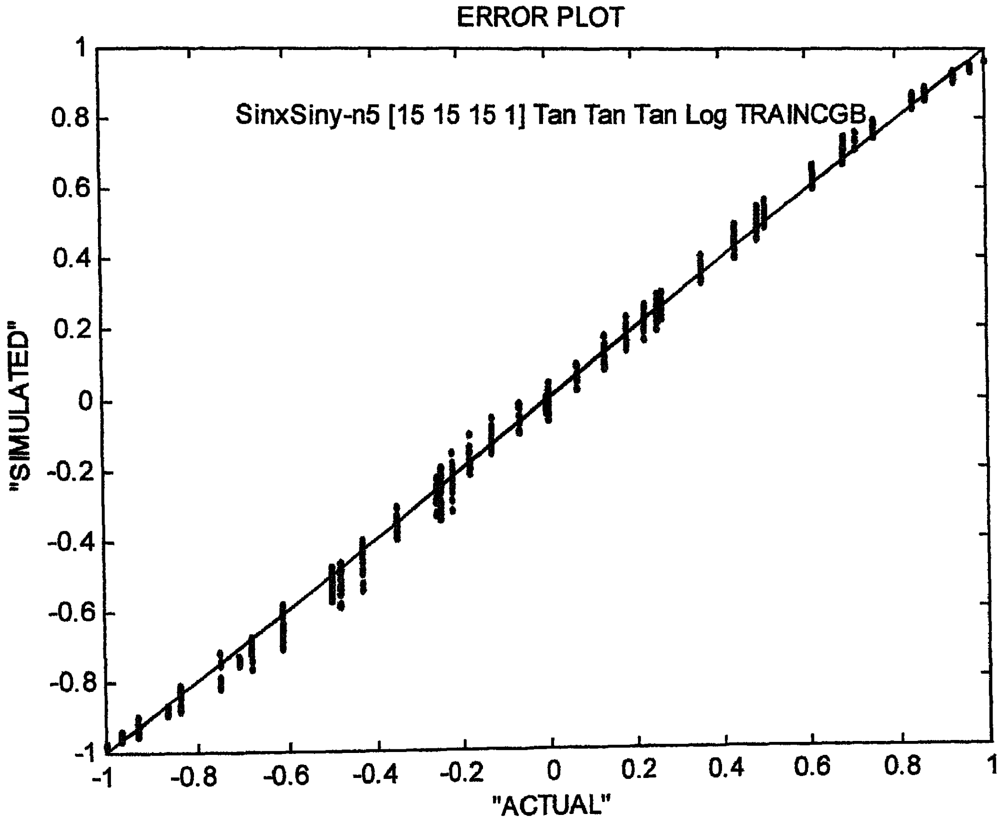
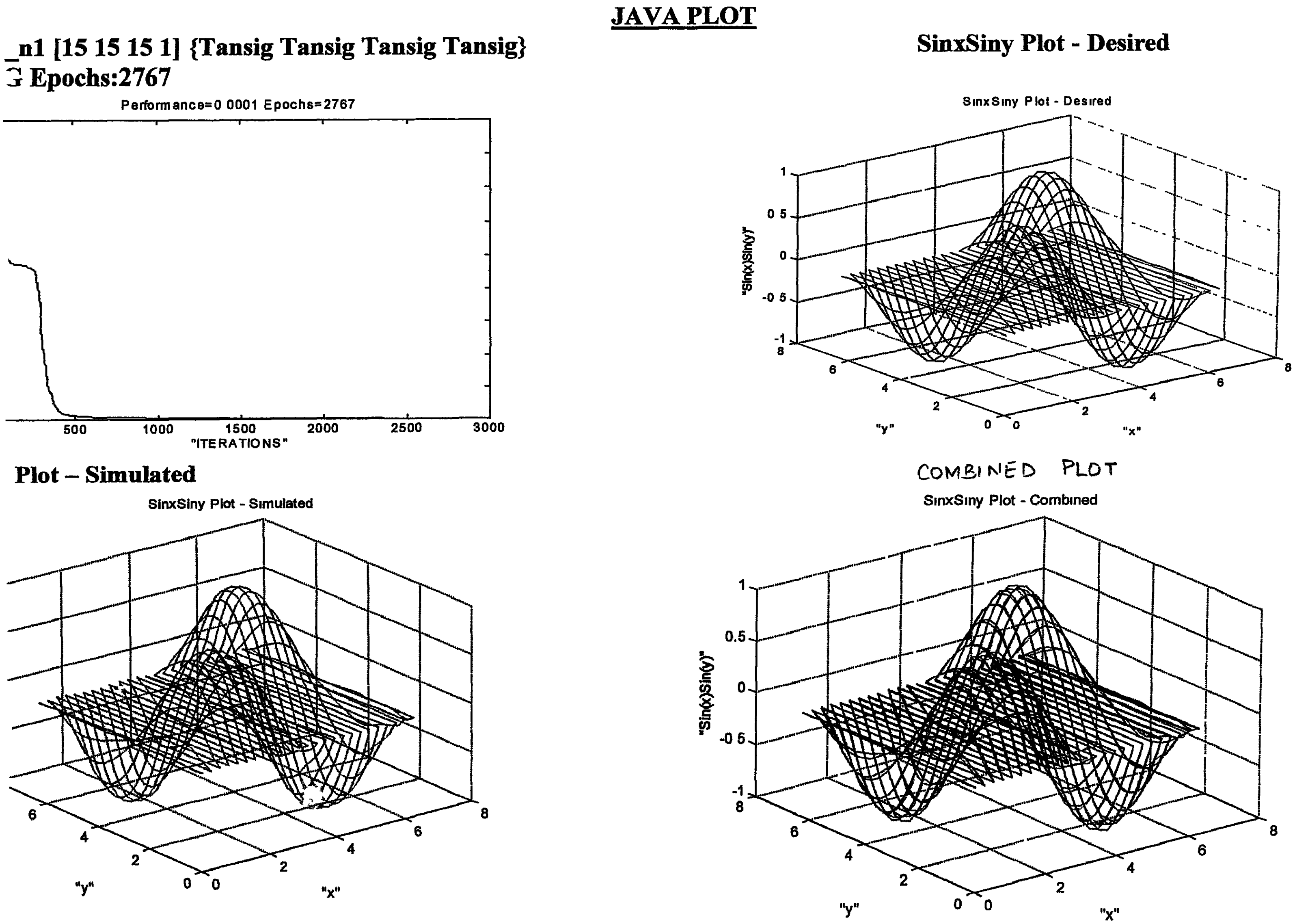


Figure 6.33d Sin(x)Sin(y) Error Plots (MATLAB)



JAVA PLOT

_n1 [15 15 15 1] {Tansig Tansig Tansig Tansig}
G Epochs:2767

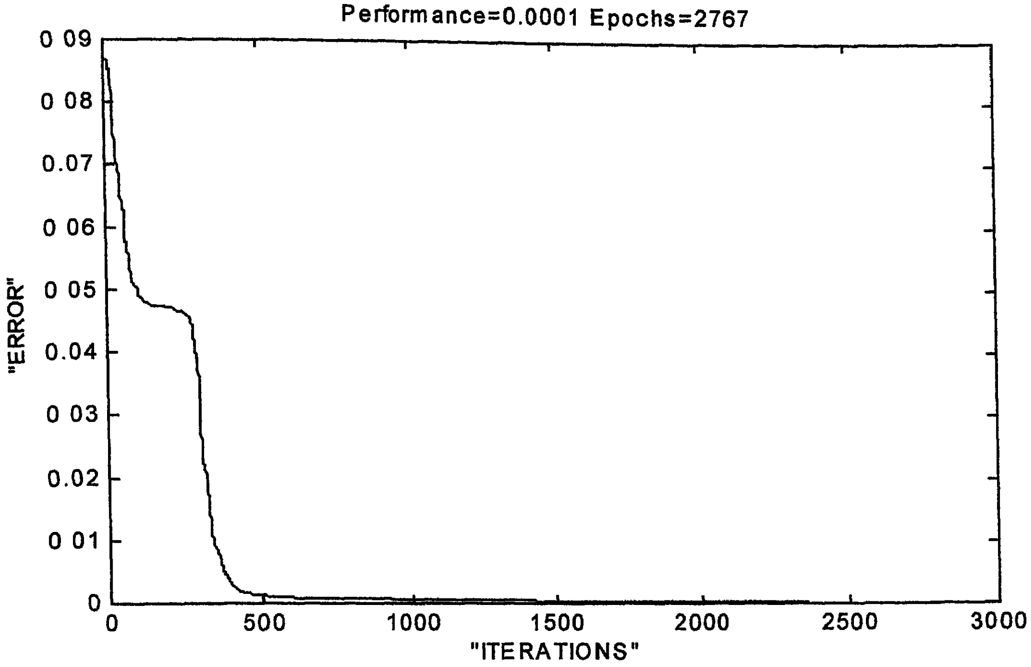
SinxSiny Plot - Desired

Plot – Simulated

COMBINED PLOT

Figure 6.34 : Sin(x)Sin(y) Plot

JAVA PLOT
SinxSiny_n1 [15 15 15 1] {Tansig Tansig Tansig Tansig}
TrainSCG Epochs:2767



SinxSiny Plot - Desired

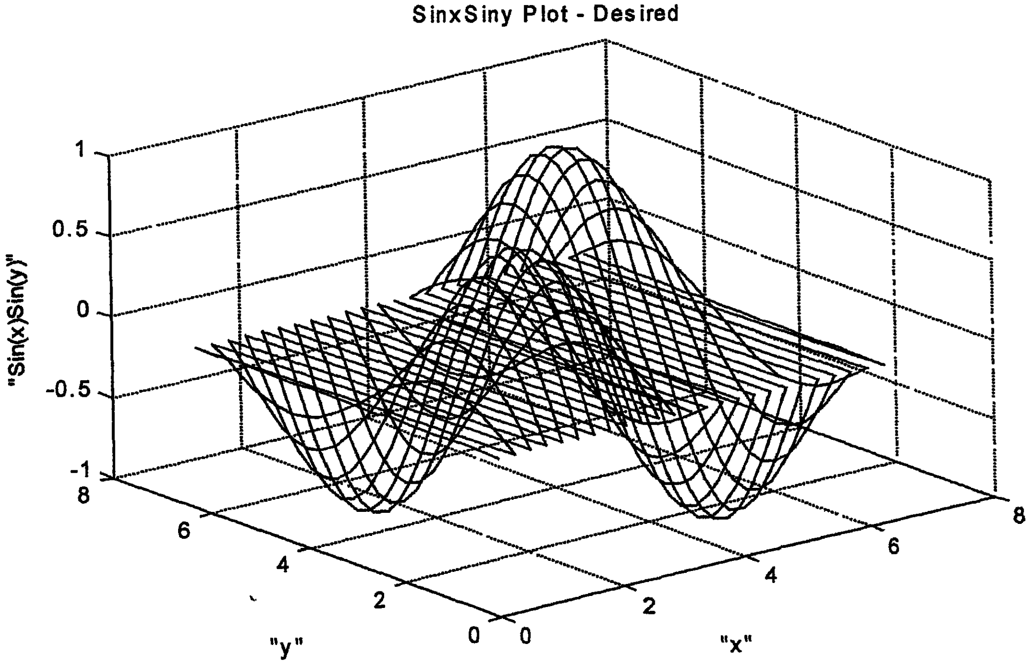
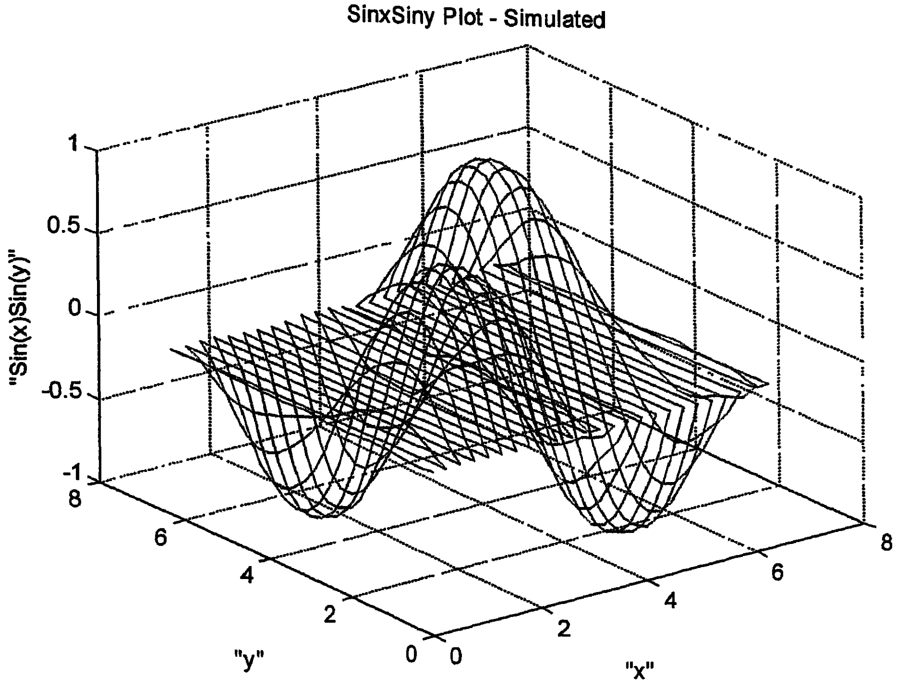


Figure 6.35: Sin(x)Sin(y) Plot

JAVA PLOT

SinxSiny_n1 [15 15 15 1] {Tansig Tansig Tansig Tansig}

SinxSiny Plot – Simulated



SinxSiny Plot – Combined

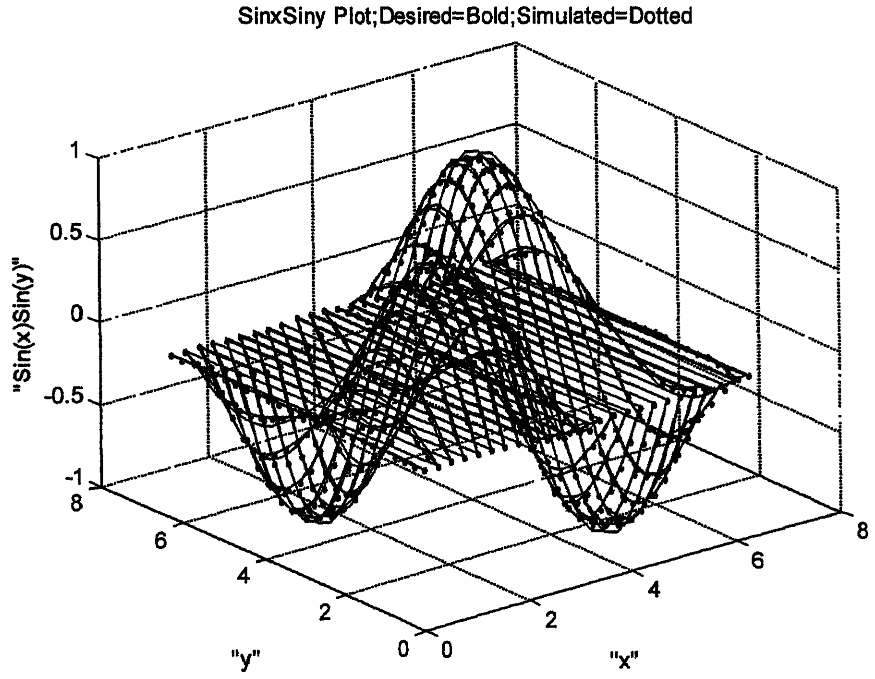
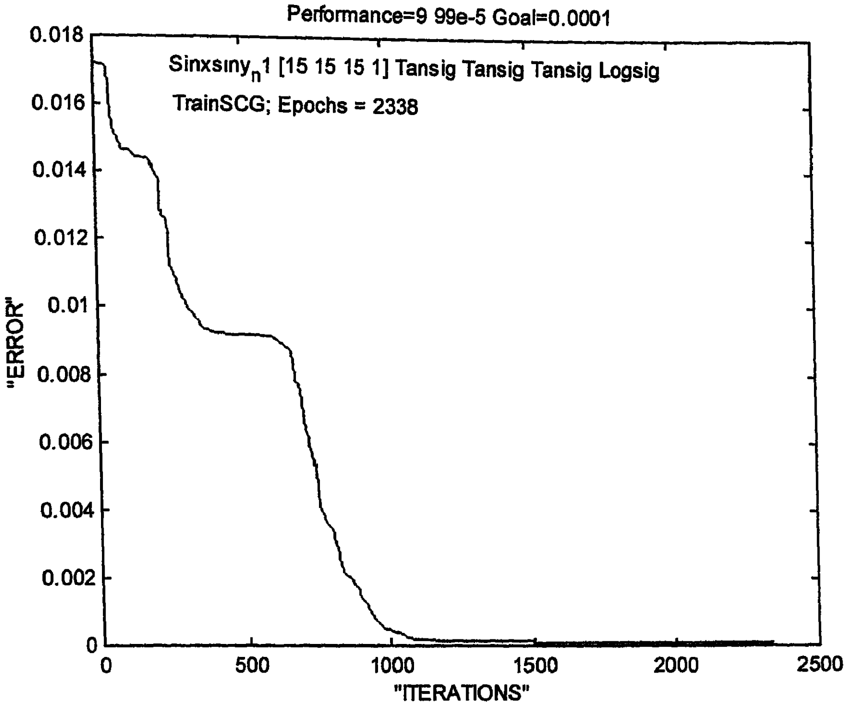


Figure 6.36 : Sin(x)Sin(y) Plot

JAVA PLOT

SinxSiny_n1[Error plot



SinxSiny Plot – Desired

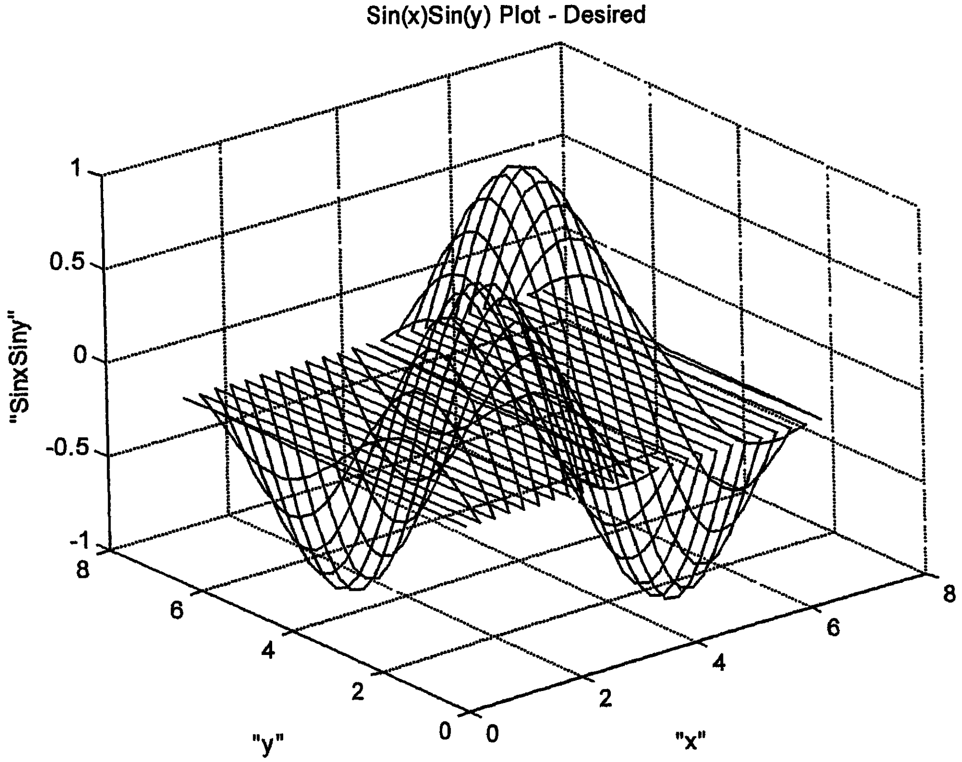
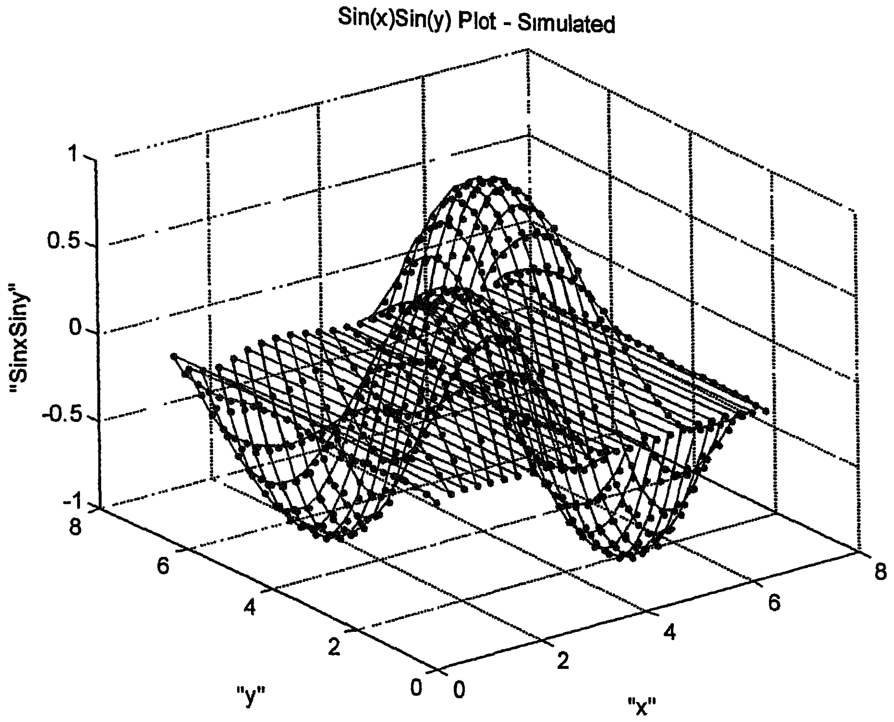


Figure 6.37: Sin(x)Sin(y) Plots

Sinxsiny n1l

SinxSiny Plot – Simulated



SinxSiny Plot – Combined
Desired = Bold(Blue); Simulated = Dotted(Red)

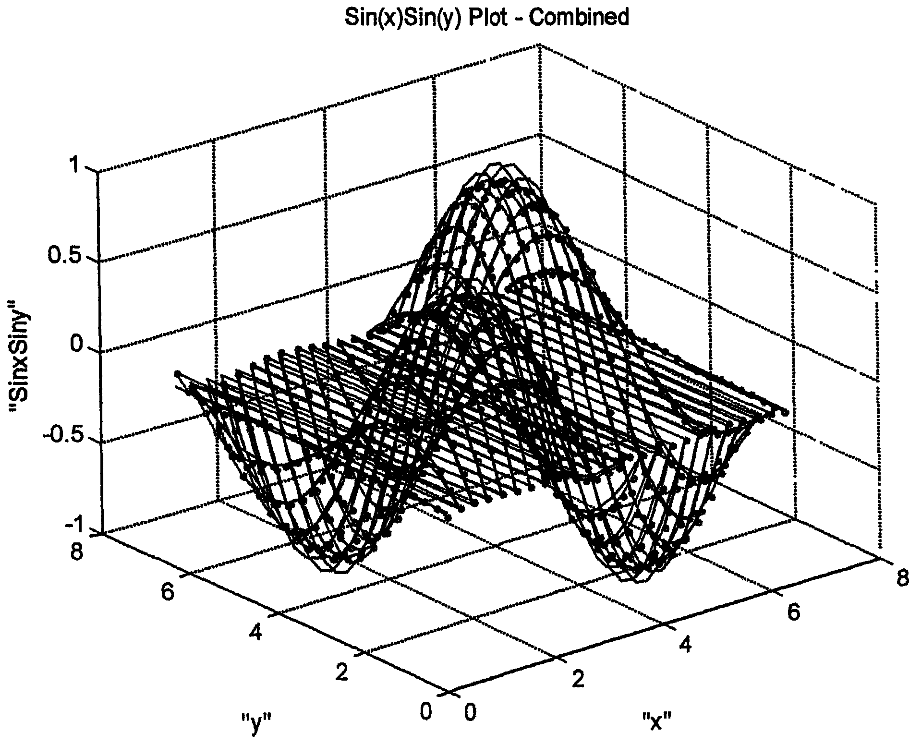
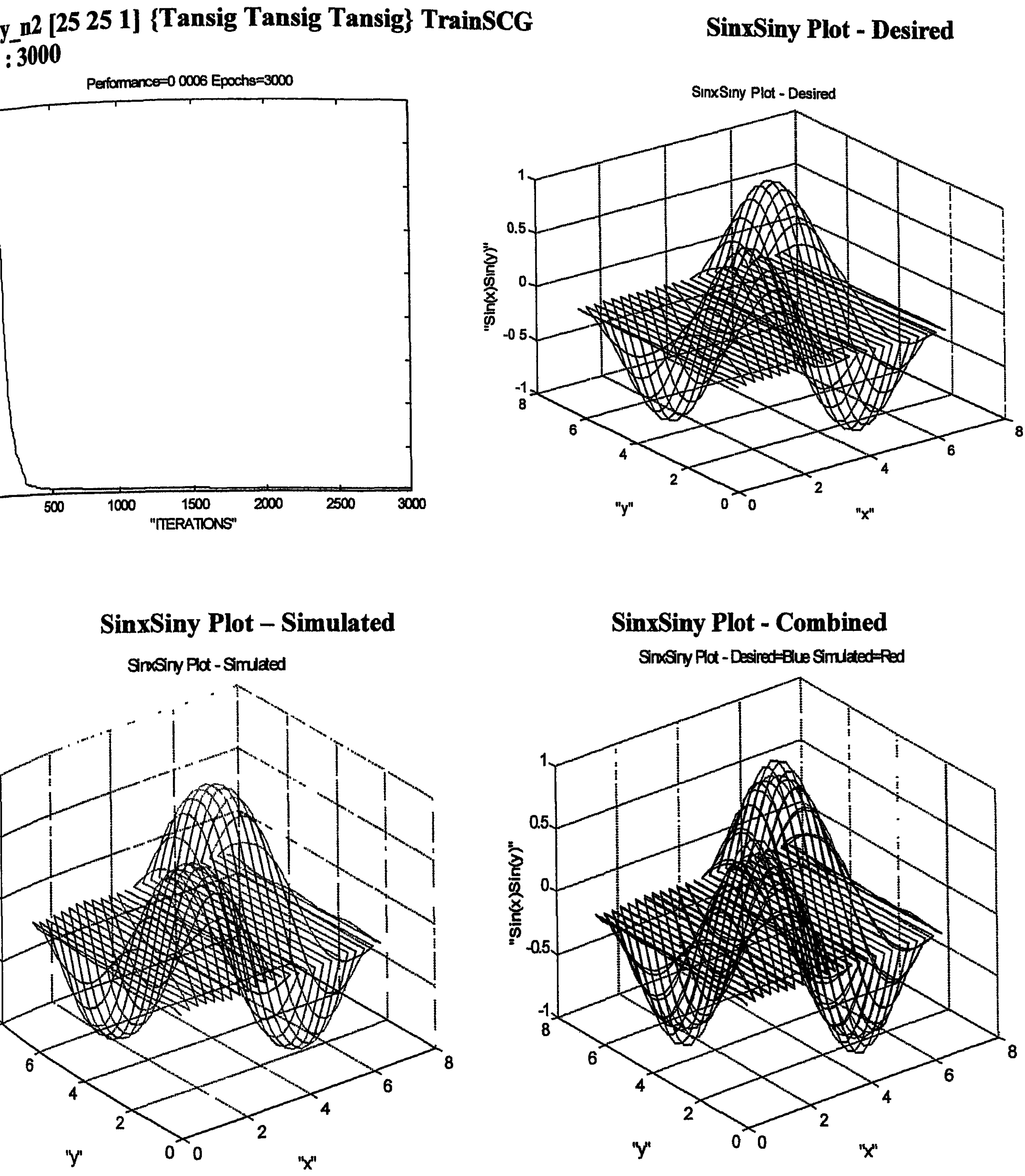


Figure 6.38: Sin(x)Sin(y) Plots

JAVA PLOT



y_n2 [25 25 1] {Tansig Tansig Tansig} TrainSCG
: 3000

SinxSiny Plot - Desired

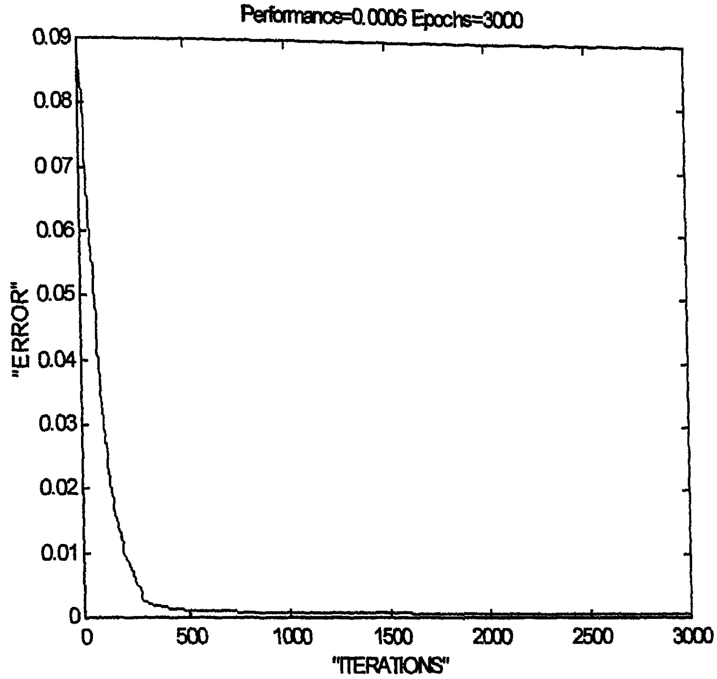
SinxSiny Plot – Simulated

SinxSiny Plot - Combined

Figure 6.39 : Sin(x)Sin(y) Plot

JAVA PLOT

Sinxsiny_n2 [25 25 1] {Tansig Tansig Tansig} TrainSCG
Epochs : 3000



SinxSiny Plot - Desired

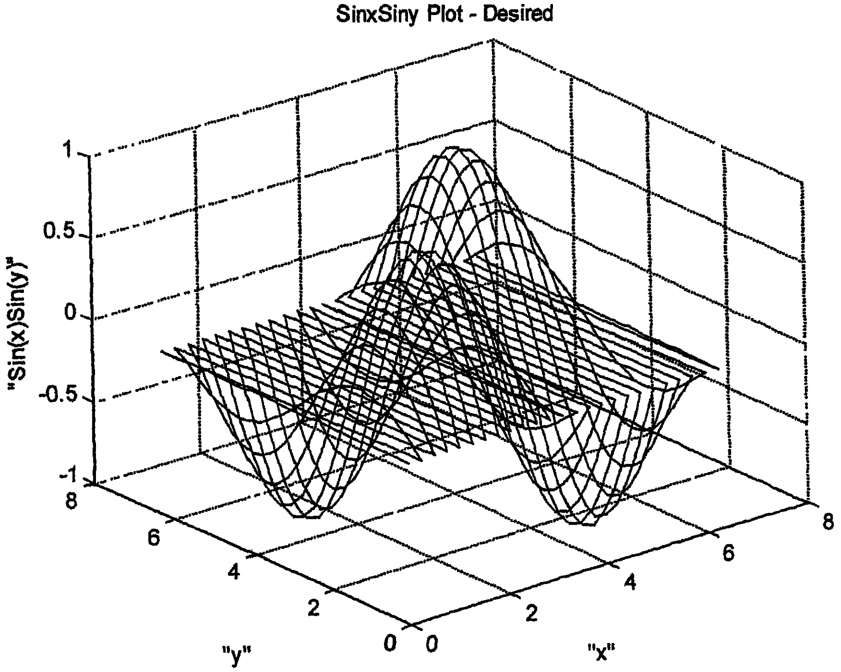
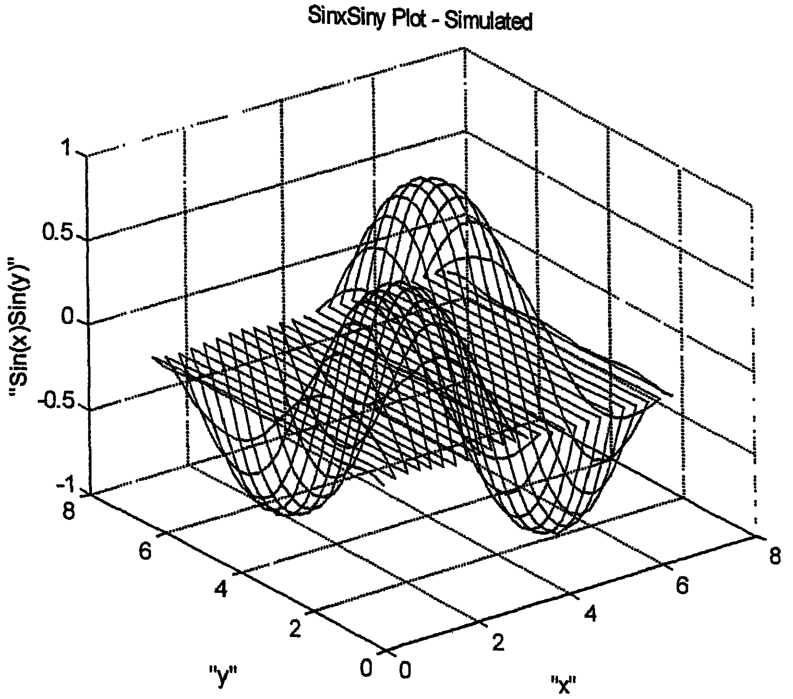


Figure 6.4ⓐ : Sin(x)Sin(y) Plot

JAVA PLOT

Sinxsiny_n2 [25 25 1] {Tansig Tansig Tansig} TrainSCG
SinxSiny Plot – Simulated



SinxSiny Plot - Combined

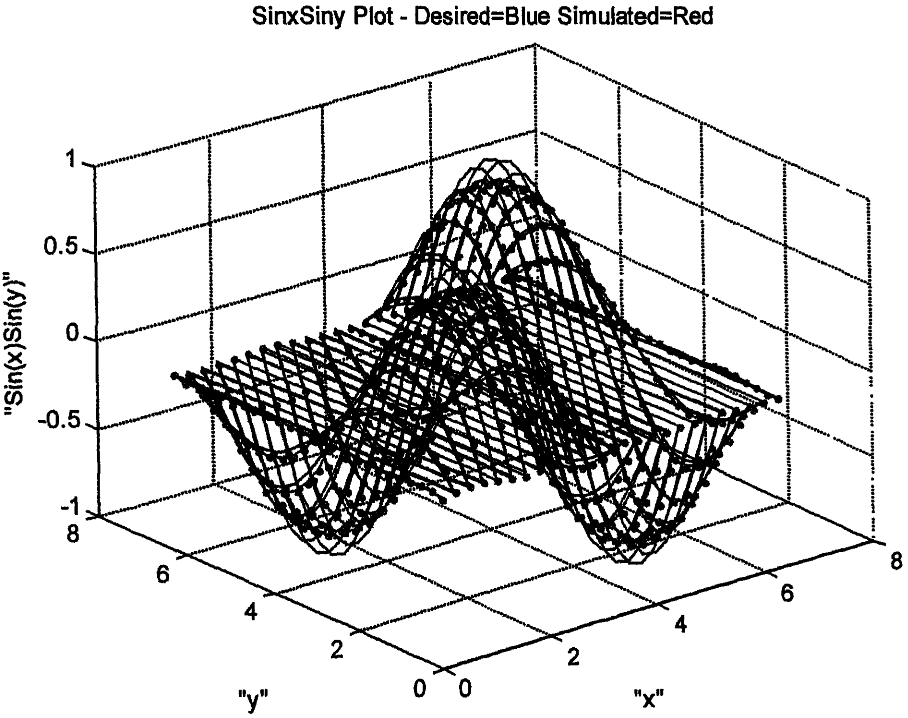
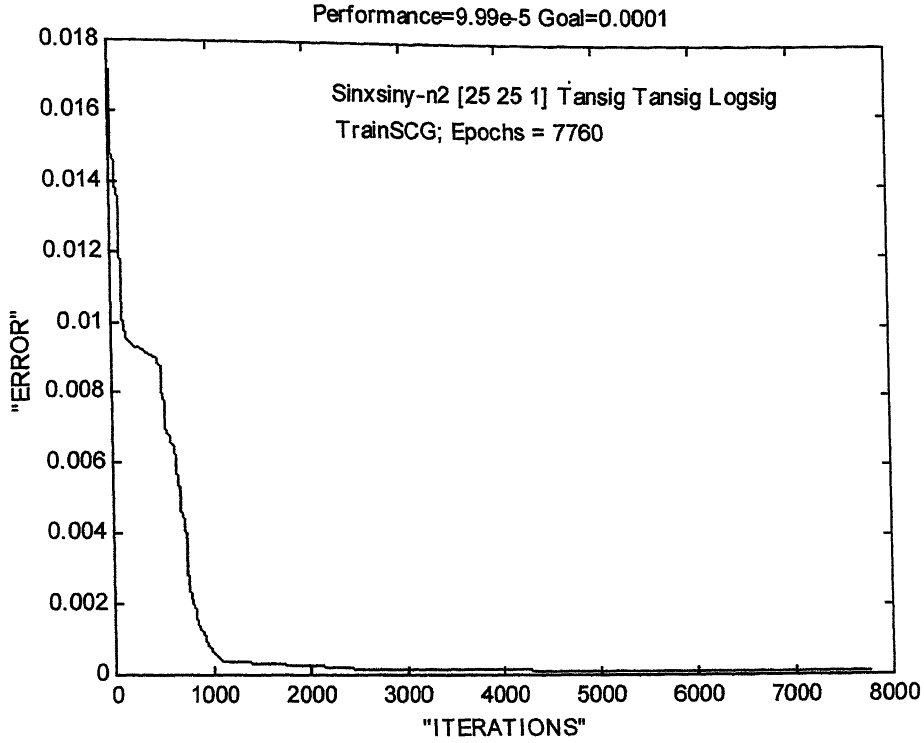


Figure 6.41 : Sin(x)Sin(y) Plot

JAVA PLOT

Sinxsiny_n2/Error Plot



Sinxsiny Plot

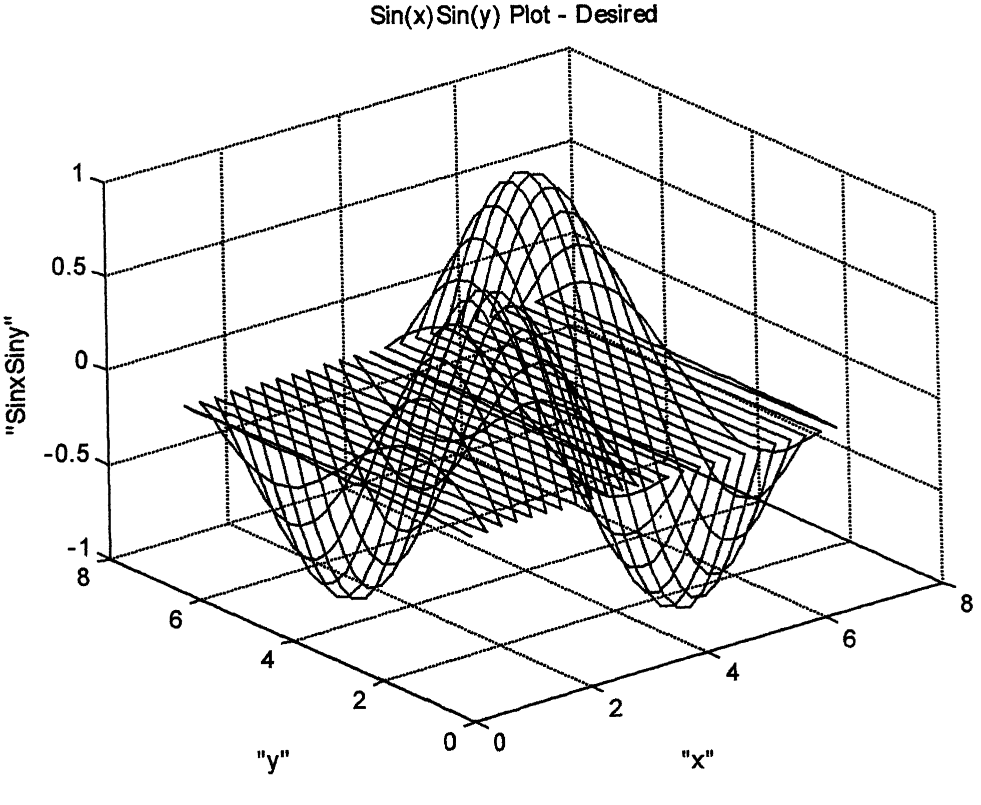
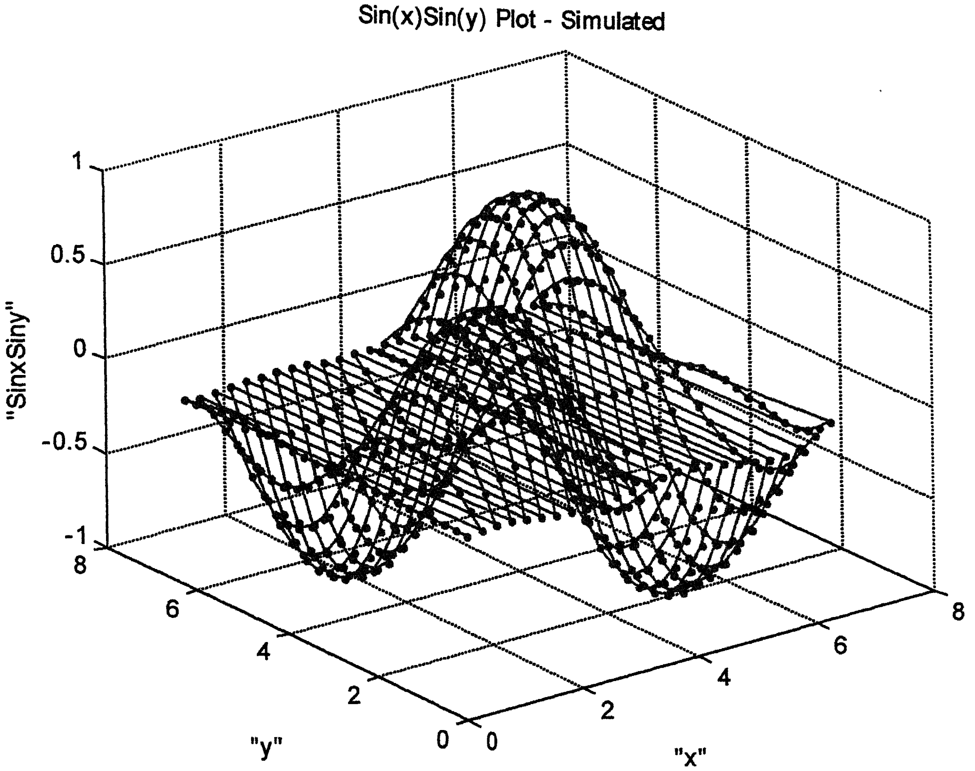


Figure 6.42: Sin(x)Sin(y) Plots

JAVA PLOT

SinxSiny_n21

SinxSiny Plot – Simulated



SinxSiny Plot – Combined
Desired = Bold(Blue); Simulated = Dotted(Red)

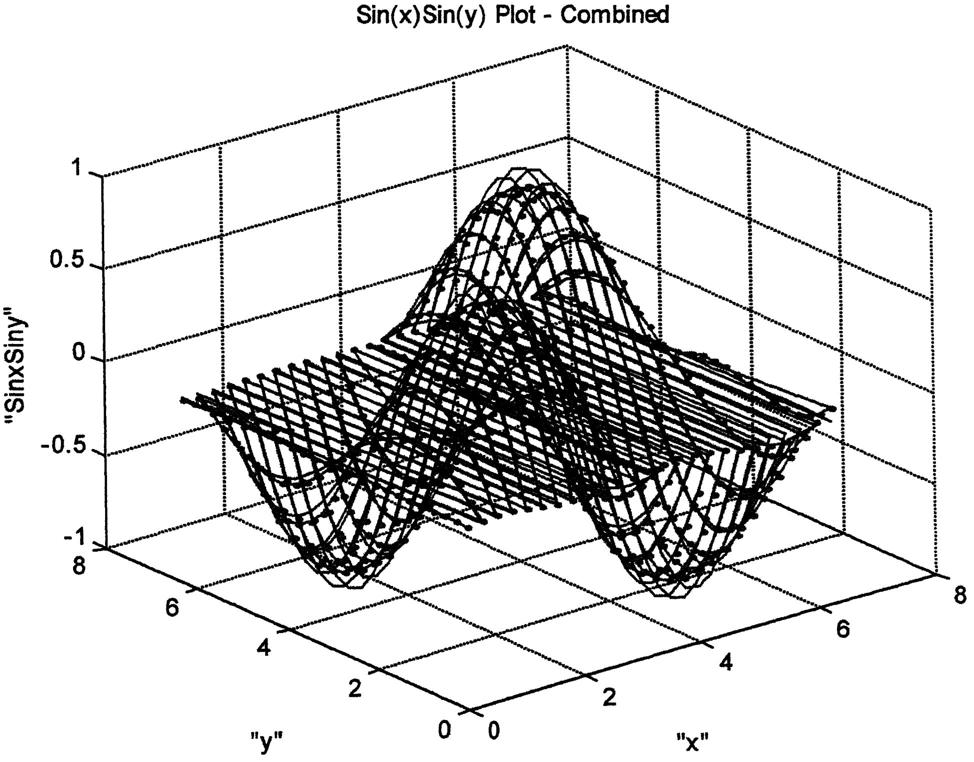
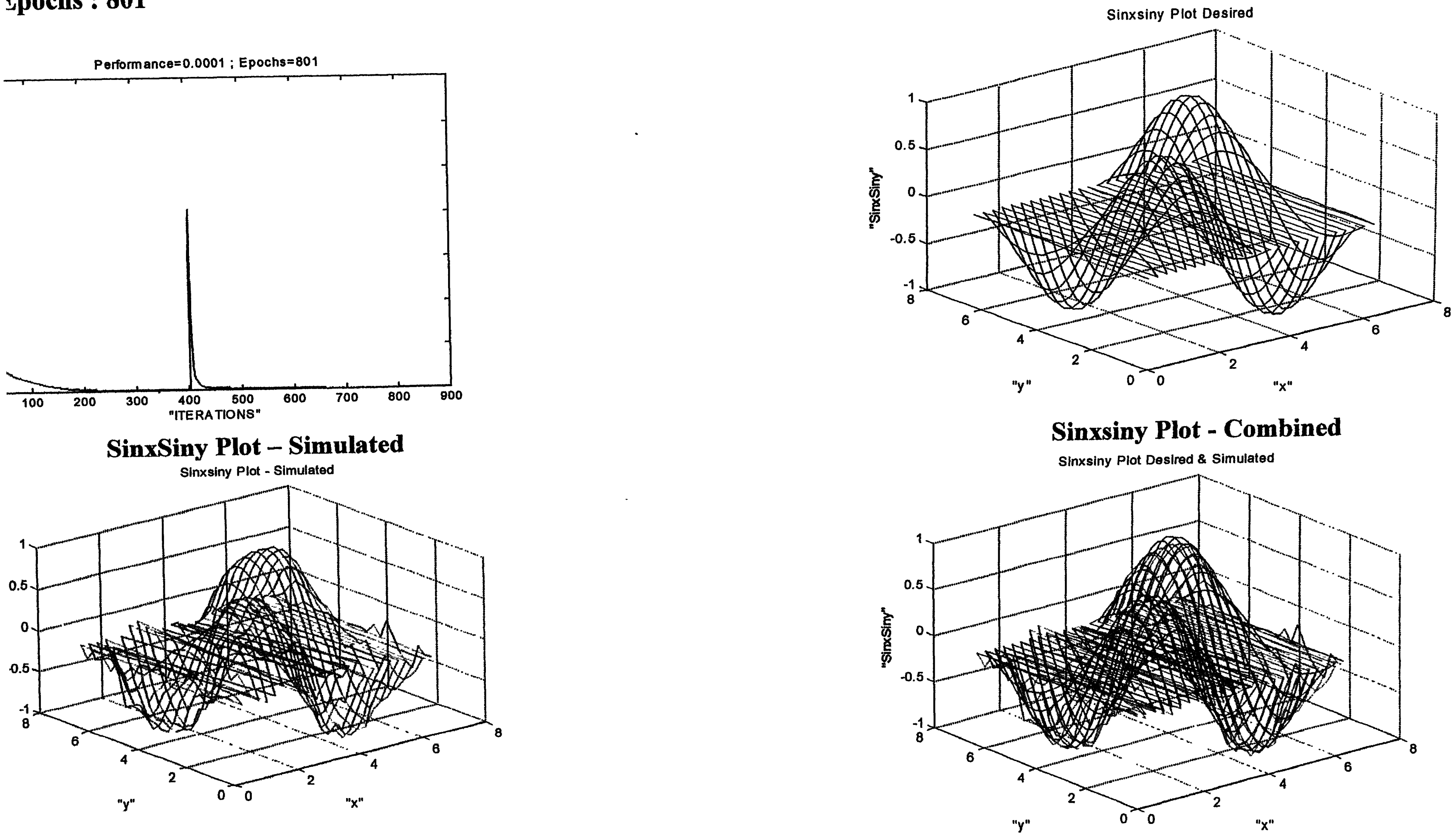


Figure 6.43: Sin(x)Sin(y) Plots

ı3 [20 25 30 1] {Tansig Tansig Tansig Logsig}
Ǝpochs : 801


SinxSiny Plot - Desired

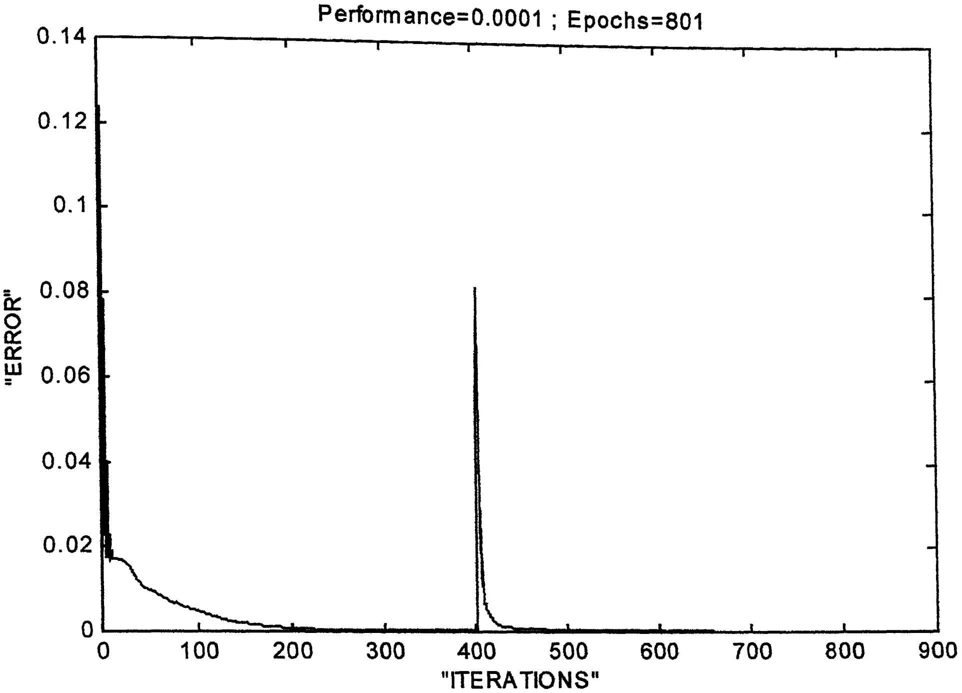
SinxSiny Plot – Simulated

Sinxsiny Plot - Combined

Figure6.44: Sin(x)Sin(y) Plot

JAVA PLOT

Sinxsiny_n3 [20 25 30 1] {Tansig Tansig Tansig Logsig} TrainRP Epochs : 801



SinxSiny Plot - Desired

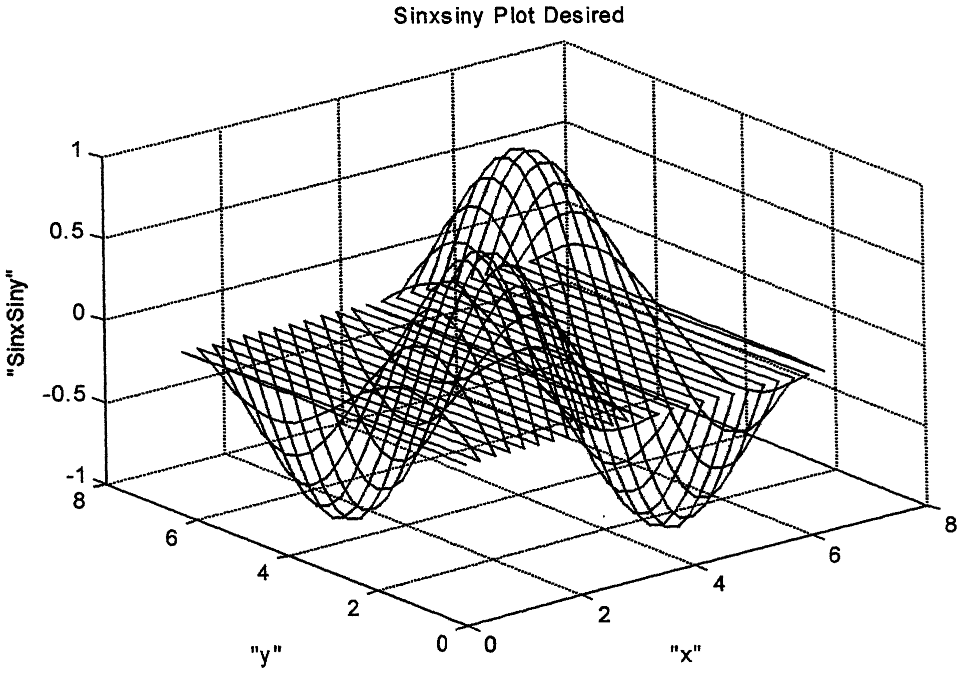
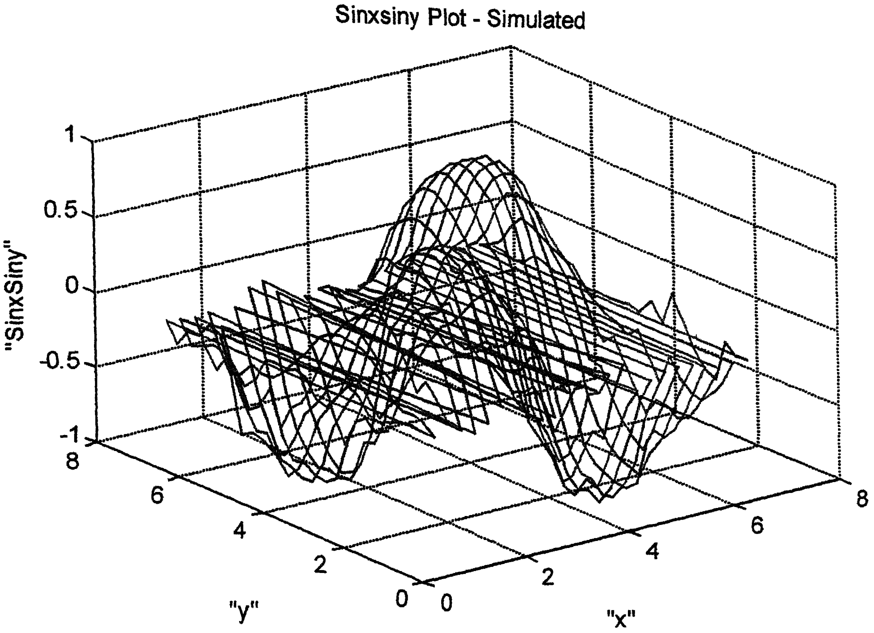


Figure6.45 : Sin(x)Sin(y) Plot

JAVA PLOT.

SinxSiny Plot – Simulated _n3



SinxSiny Plot – Combined

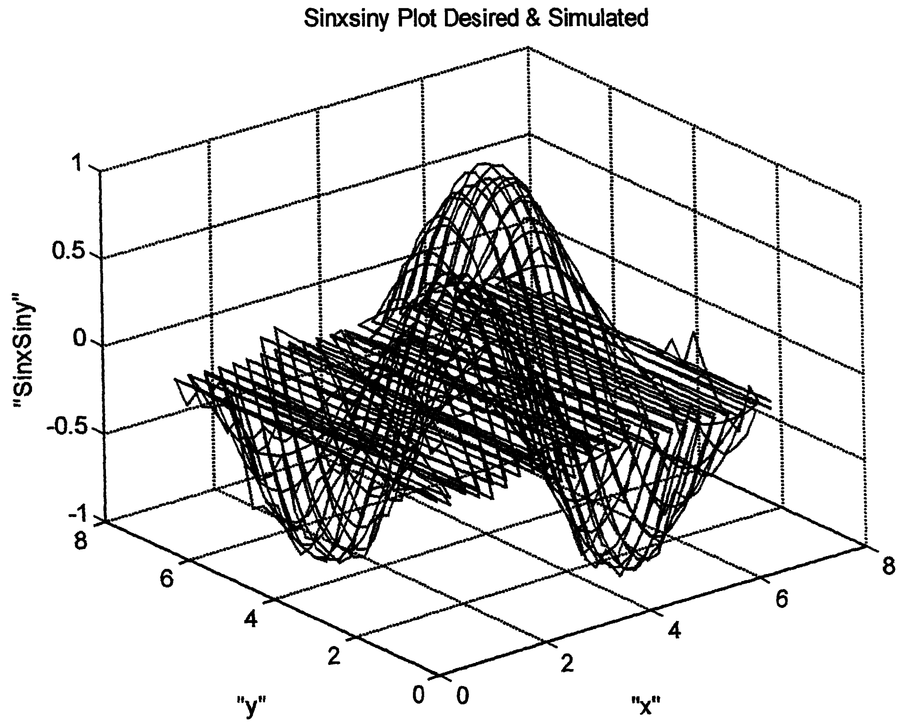
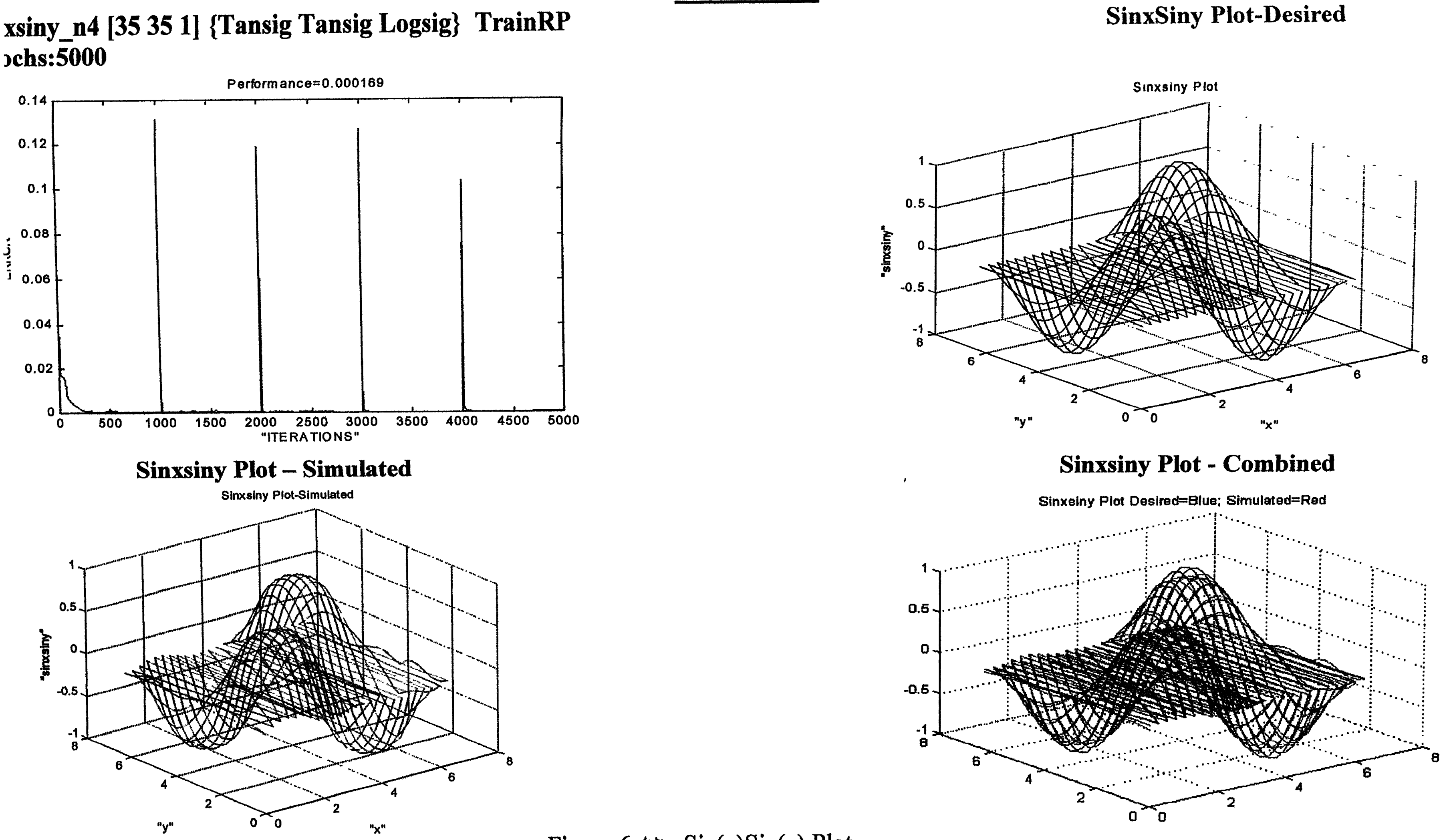


Figure 6.46 :Sin(x)Sin(y) Plot

JAVA PLOT



xsiny_n4 [35 35 1] {Tansig Tansig Logsig} TrainRP
ochs:5000

SinxSiny Plot-Desired

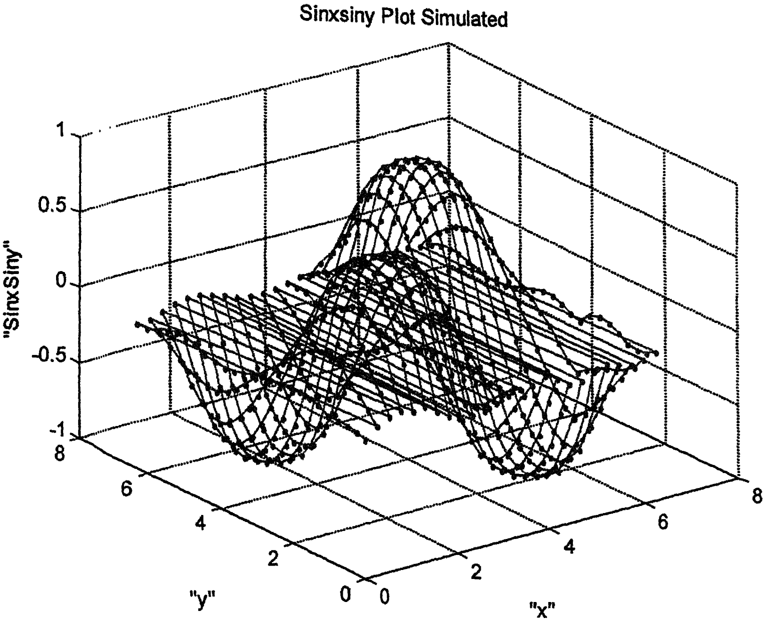
Sinxsiny Plot – Simulated

Sinxsiny Plot - Combined

Figure 6. 47 : Sin(x)Sin(y) Plot

JAVA PLOT

SinxSiny Plot-Simulated JA4



SinxSiny Plot – Combined

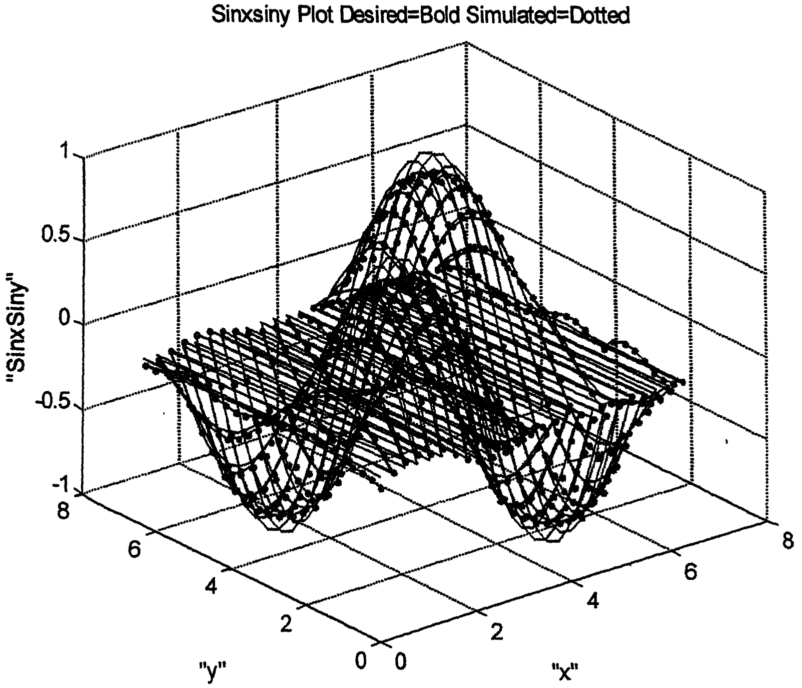


Figure 6.49:Sin(x)Sin(y) Plot

Sin(x)Sin(y)-Error Plots (JAVA)

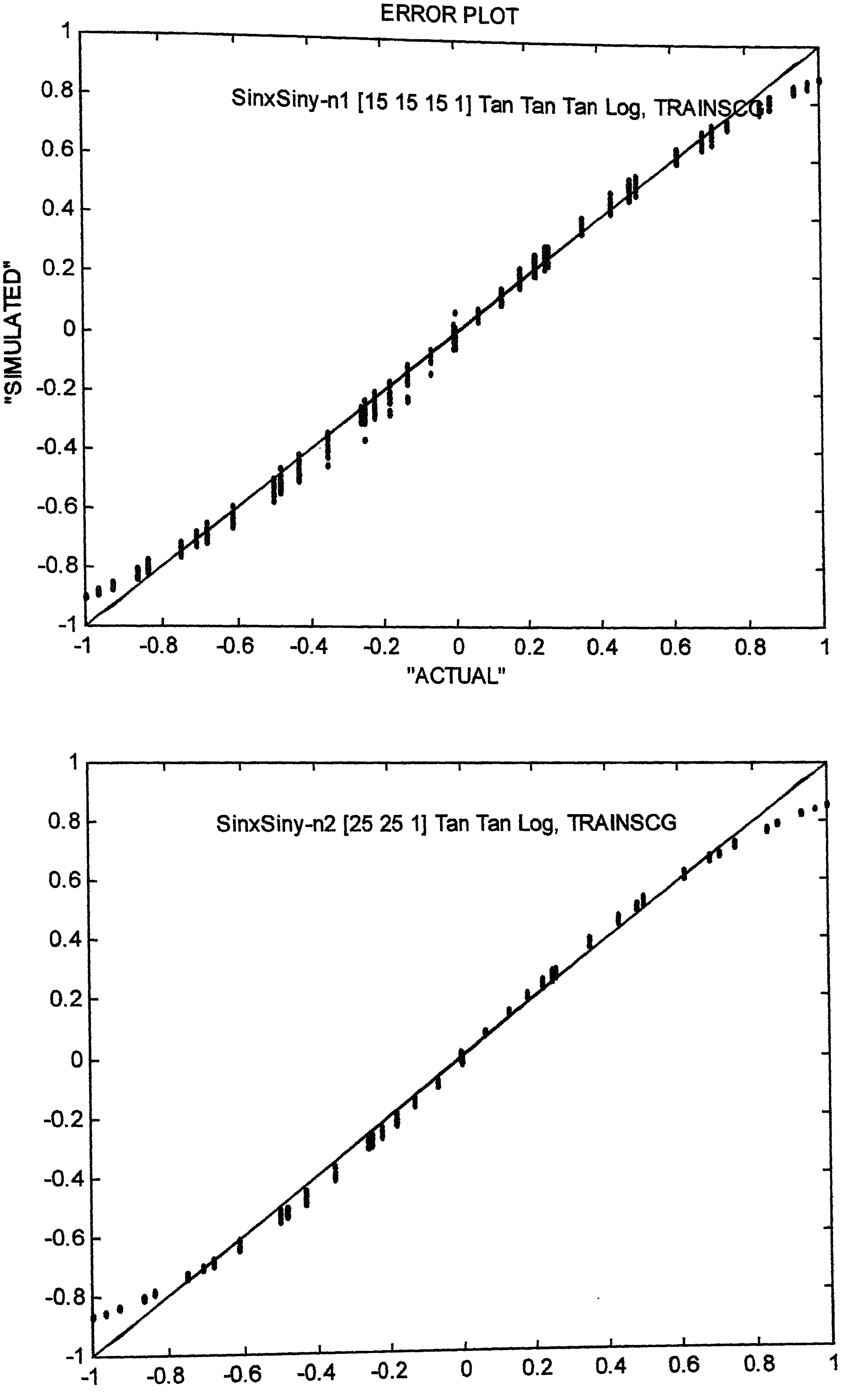


Figure 6.49:Sin(x)Sin(y) Error Plots (JAVA)

Sin(x)Sin(y)-Error Plots (JAVA)

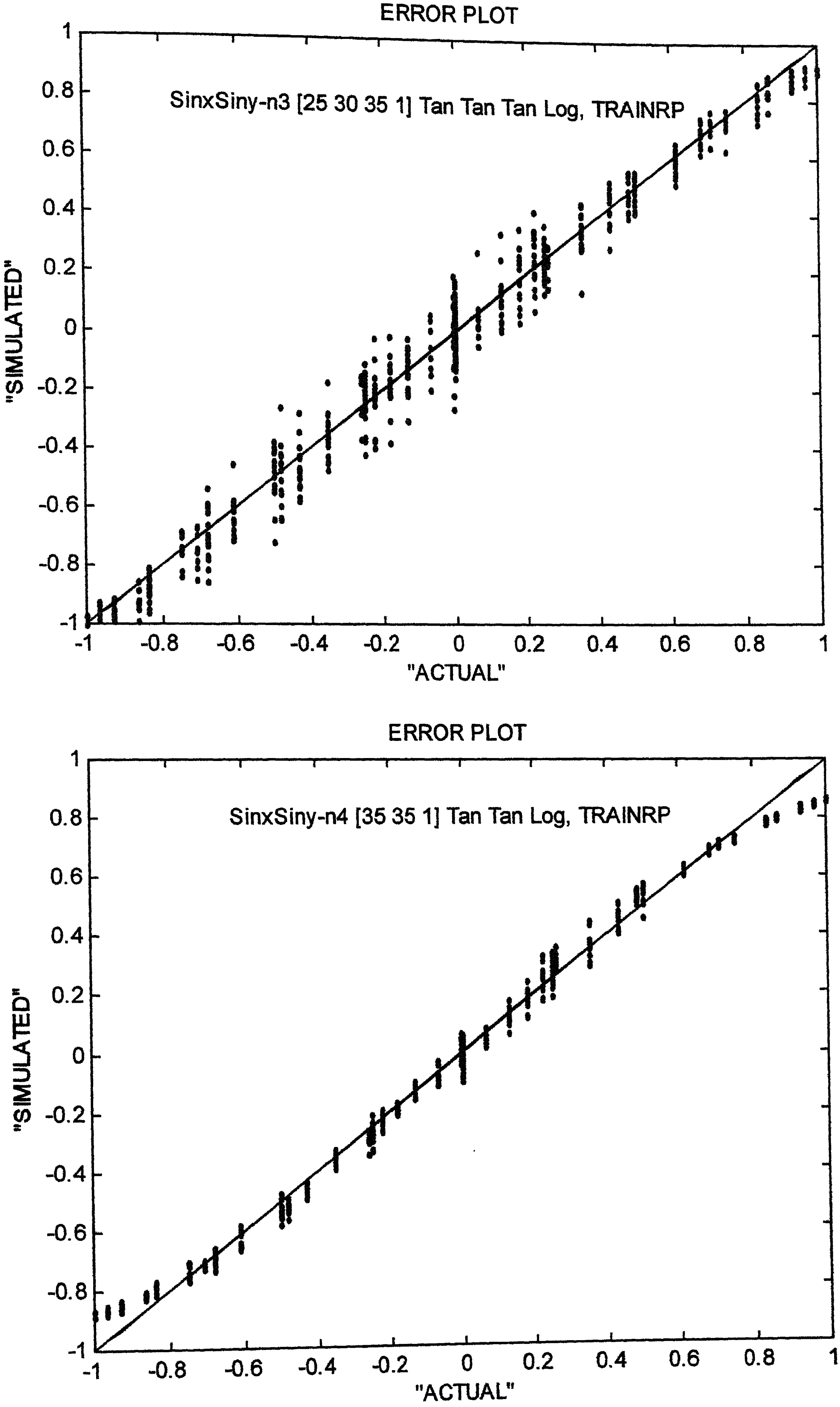


Figure 6.4c: Sin(x)Sin(y) Error Plots (JAVA)

6.6 Classification Problem

In the problem considered, it is required to classify points into four classes. The points are generated using two functions g1 and g2. The values of g1 and g2 decide the class in which the point should be classified. The two functions are:

$$g_1 (x_1, x_2, x_3, x_4, x_5) = x_2 * x_3 * \exp\left[(x_3 * x_4 - x_5)^2\right] - 2\left[(x_1 * x_4 - 1)^2\right] * x_5 + x_3 - x_4 - 0.4$$

$$g_2 (x_1, x_2, x_3, x_4, x_5) = x_2 * x_3 \,(2\, x_4 * x_5 - 1) - Sin\left(\frac{3}{2}\pi * x_1\right) - 1$$

The classification has been according to the rules:

If $g_1 < 0$ and $g_2 < 0$ then the class = 0 0

If $g_1 < 0$ and $g_2 > 0$ then the class = 0 1

If $g_1 > 0$ and $g_2 < 0$ then the class = 1 0

If $g_1 > 0$ and $g_2 > 0$ then the class = 1 1

The values of the functions depend on the variables x_1 to x_5, which vary from 0 to 2 at spacing of 0.2/0.3/0.4. A total of 1.6 lakh points were generated and classified. A total of 5 – 8 % of the total points generated have been used for training and the rest were used for simulation. The results and plots of classification are shown in figures 6.50 (MATLAB) and figures 5.51 to 5.53 (JAVA). The architectures that were found to give convergence for the above problem are:

MATLAB RESULTS

Table 6.12: Classification Problem

Classification Problem: ANN Architectures used; Error Goal = 0.01				
Network	Architecture	Activation Fn.	Algorithm	Iterations
Class21rp	61 59 61 2	Tansig Tansig Tansig Logsig	Trainrp	573
Class31rp	19 17 19 2	Tansig Tansig Tansig Logsig	Trainrp	>30000
Class41rp	37 35 37 2	Tansig Tansig Tansig Logsig	Trainrp	113
Class21scg	61 59 61 2	Tansig Tansig Tansig Logsig	Trainscg	753
Class31scg	21 19 21 2	Tansig Tansig Tansig Logsig	Trainscg	1332
Class41scg	15 15 15 2	Tansig Tansig Tansig Logsig	Trainscg	1192
Class42scg	61 49 45 2	Tansig Tansig Tansig Logsig	Trainscg	101

JAVA RESULTS

Table 6.13 :Classification Problem

Classification Problem: ANN Architectures used; Error Goal = 0.01

Network	Architecture	Activation Fn.	Algorithm	Iterations
Class212	61 59 61 2	Tansig Tansig Tansig Logsig	Trainrp	1814
Class31	19 17 19 2	Tansig Tansig Tansig Logsig	Trainrp	3000
Class41	37 35 37 2	Tansig Tansig Tansig Logsig	Trainrp	393
Class212	61 59 61 2	Tansig Tansig Tansig Tangsig	Trainscg	2643
Class31	21 19 21 2	Tansig Tansig Tansig Tangsig	Trainscg	2500
Class42	61 49 45 2	Tansig Tansig Tansig Tangsig	Trainscg	646
Class212	61 59 61 2	Tansig Tansig Tansig Logsig	Trainscg	2761
Class31	21 19 21 2	Tansig Tansig Tansig Logsig	Trainscg	1645
Class41	15 15 15 2	Tansig Tansig Tansig Logsig	Trainscg	961

On simulation of the trained networks it was found that the error in classification was around 6 - 8 % in the case of Trainrp algorithms and around 4– 6% in the case of Trainscg for MATLAB.In the case of JAVA the error performance was slightly better.

Conclusion: From the above results it is concluded that:

(a) Trainrp algorithm gave better results in the case of JAVA.

(b) The minimum error in classification was achieved when trainscg was used.

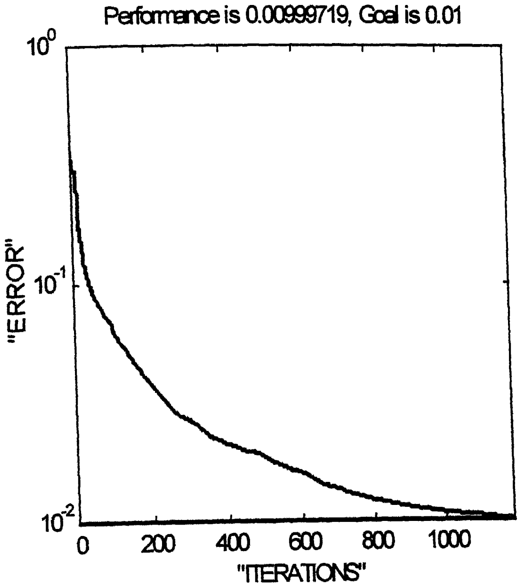
(c) The use of points near the boundaries for training improved the network performance.

(d) Around 7 to 10% points are required for training.

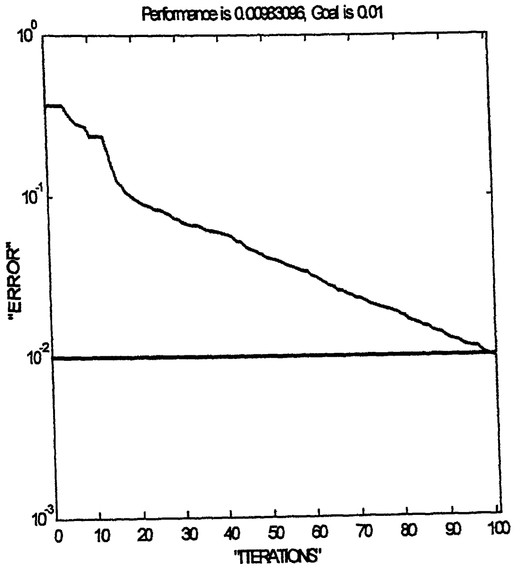
(e) The network required at least three hidden layers.

(f) The number of neurons required were large and hence training takes considerable time.

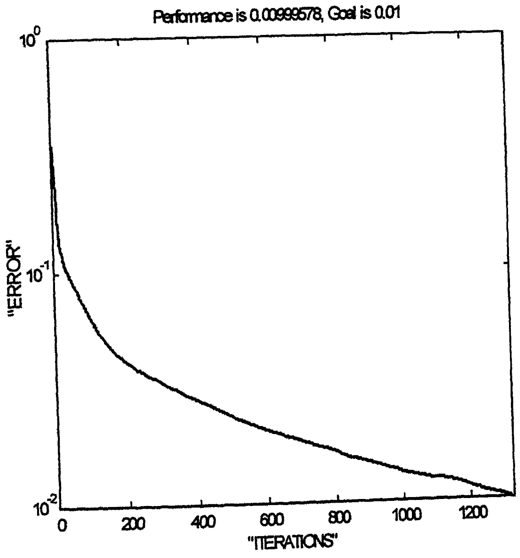
Class41scg
[15 15 15 2]{Tan Tan Tan Log}
Trainscg Epochs: 1192



Class42scg
[61 49 45 2]{Tan Tan Tan Log}
Trainscg Epochs : 101



Class31scg
[21 19 21 2]{Tan Tan Tan Log}
Trainscg Epochs : 1332



Class 21scg
[61 59 61 2] {Tan Tan Tan Log}
Trainscg Epochs : 743

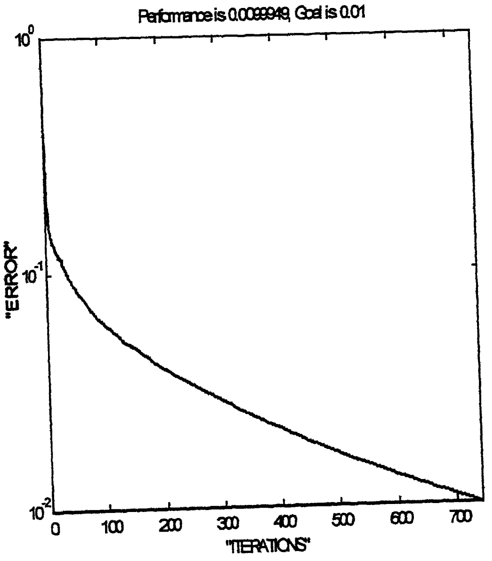
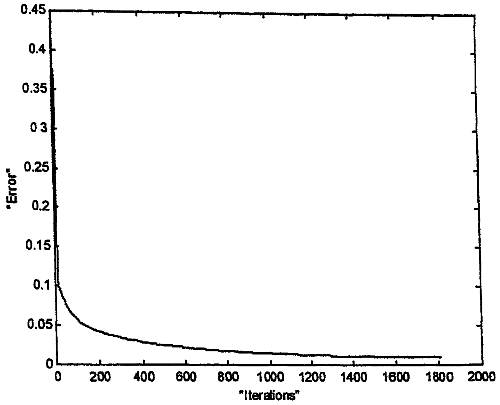


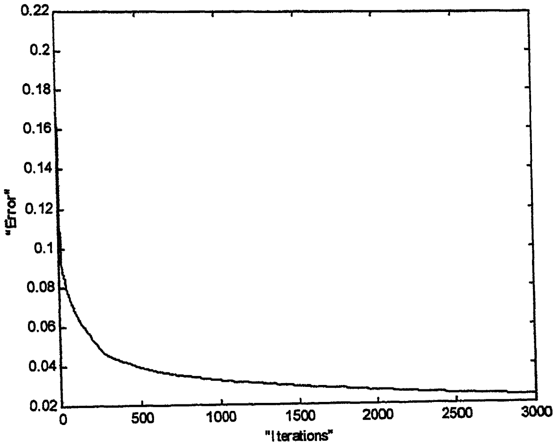
Figure6.5c: Classification Plots

JAVA PLOT

Class 212 [61 59 61 2] { Tansig Tansig Tansig Logsig} Trainrp Epochs: 1814



Class 31 [19 17 19 2] { Tansig Tansig Tansig Logsig} Trainrp Epochs: 3000



Class 41 [37 35 37 2] { Tansig Tansig Tansig Logsig} Trainrp Epochs:393

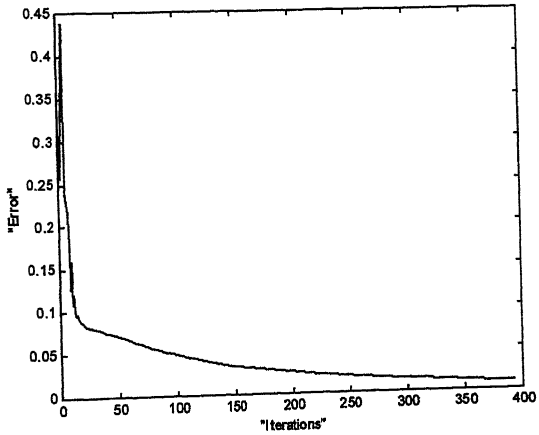
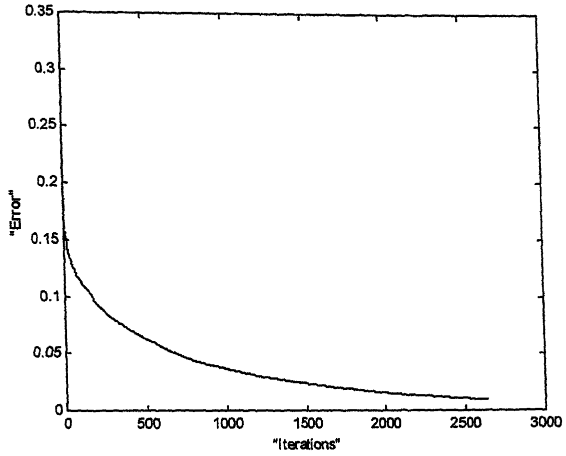


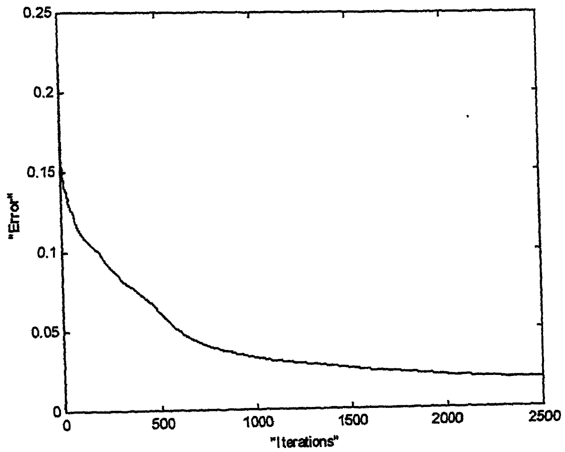
Figure 657 : Classification - Error Plots

JAVA PLOTS

Class212 : [61 59 61 2] { Tansig Tansig Tansig Tansig} Trainscg Epochs:2643



Class31 : [21 19 21 2] { Tansig Tansig Tansig Tansig} Trainscg Epochs:2500



Class42 : [61 49 41 2] { Tansig Tansig Tansig Tansig} Trainscg Epochs:646

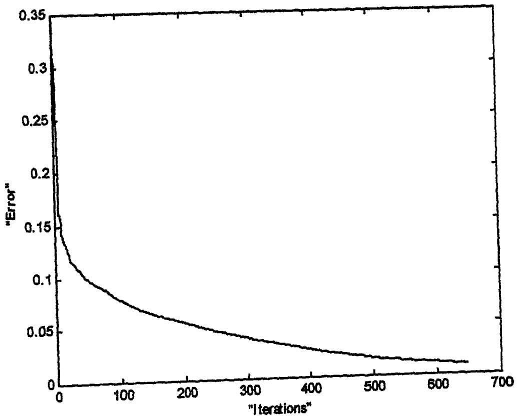
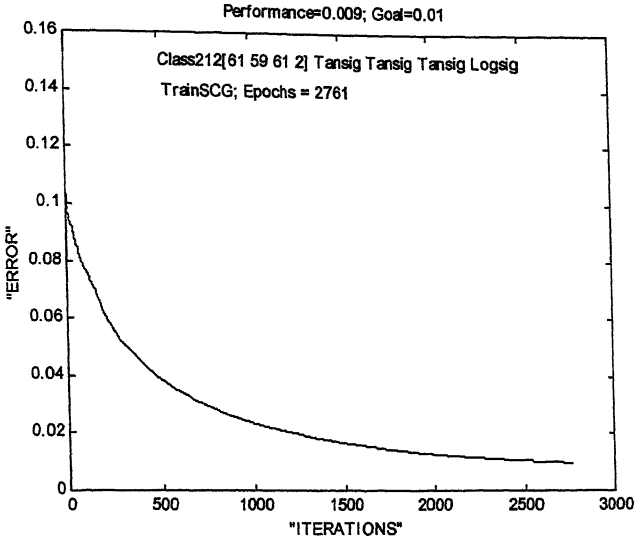


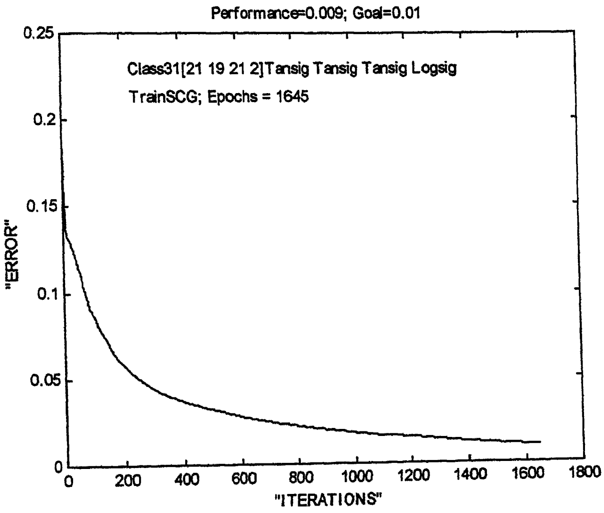
Figure 6.52: Classification Problem

JAVA PLOT

Class212-Classification problem



Class31



Class41

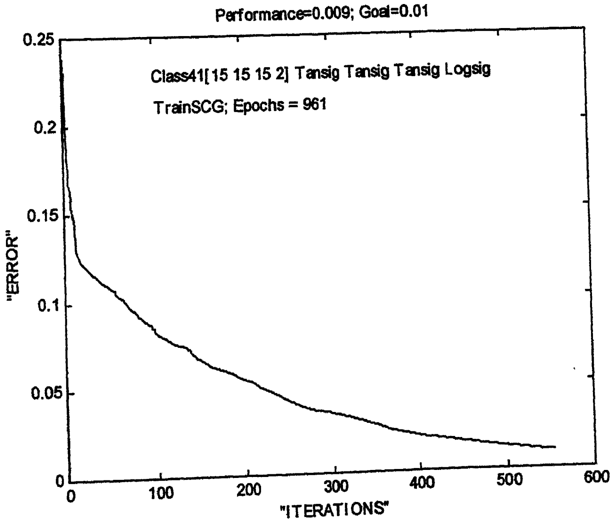


Figure 6.5b: Classification Problem Error Plots

6.7 Exponential Sine Curve Problem

In this problem time series analysis was carried out. This is a problem of function approximation. The function is an exponentially increasing sine curve. The effort was to build a network that could predict the future points of the curve, given the previous ones. The problem was solved using three networks. One each for sinusoidal and exponential curves and the third one for the final output (sin(x)exp(x)). In the individual networks some of the points were used for training and the remaining were generated or predicted. Both these networks used time series prediction. Every three consecutive input-output sets were used for prediction of the fourth output. This is how the training set was made. On training the individual networks were used to predict the remaining of the curve. The third network was employed to carry out the operation of multiplication. The inputs to this network are the predicted sine (sin(x)) and exponential (exp(x)) points and the output is sin(x)*exp(x). Approximating a multiplication operation of such a kind required a very thorough search for an appropriate network. The results and the plots are shown in figures 6.54 to5.57. Two networks used, one each for sinusoidal and exponential curve time series prediction and one for the prediction of output. Z(x,y) = Sin(x) Sin(y). . The network architectures that were found to give good approximation are:

MATLAB RESULTS

Table 6.14: Sin(x)Exp(x) Approximation

Sinexp Function Approximation Problem: ANN Architectures used;					
Network	Architecture	Activation Fn.	Algorithm	Epochs	Error Goal
Exp_fn	Single neuron	Purelin	Trainrp	229	1 e-7
Sine_fn	Single neuron	Purelin	Trainrp	800	1 e-7
Sinexp_fn	[21 20 21 1]	Tansig Tansig Tansig Tansig	Trainrp	3390	7 e-7
Exp_fn1	Single neuron	Purelin	Trainscg	17	1 e-7
Sine_fn1	Single neuron	Purelin	Trainscg	6	1 e-7
Sinexp_fn1	[21 20 21 1]	Tansig Tansig Tansig Tansig	Trainrp	28538	7 e-7

JAVA RESULTS

Table 6.15: Sin(x)Exp(x) Approximation

Sinexp Function Approximation Problem: ANN Architectures used;					
Network	Architecture	Activation Fn.	Algorithm	Iterations	Error Goal
Exp_fn	Single neuron	Purelin	Trainrp	100000	2 e-3
Sine_fn	Single neuron	Purelin	Trainrp	347	4 e-5
Sinexp_fn	[21 20 21 1]	Tansig Tansig Tansig Tansig	Trainrp	97466	1 e-6

Conclusion: From the above results it can be concluded that:

(a) Only Trainrp and Trainscg could perform the time series prediction of sine and exponential curves.

(b) The error goal reached by MATLAB networks was more rigorous. However, even with inferior error performance JAVA predicted the curve well.

(c) JAVA codes took sufficiently long time to converge.

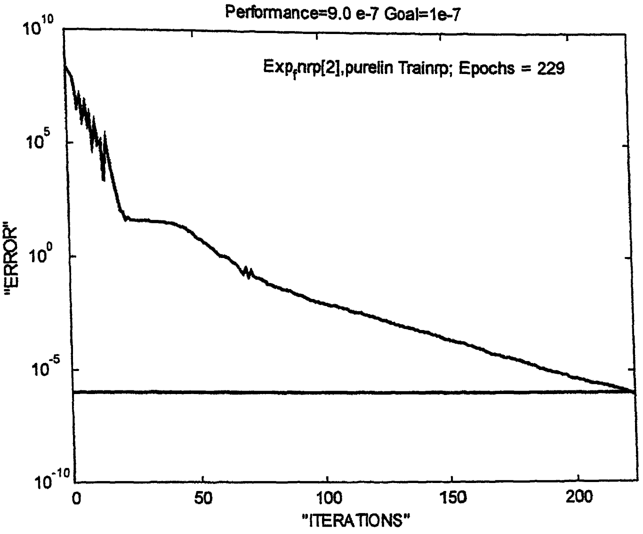
(d) Time series prediction for sine and exponential curves requires a single neuron in the output layer.

(e) To perform the function of multiplication neural network requires a very big architecture, with as many as three hidden layers having sixty neurons.

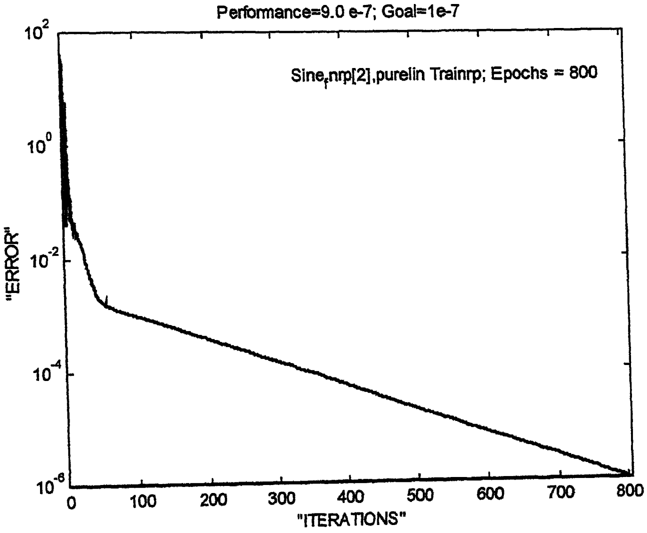
(f) Time series prediction using Trainscg does not work in JAVA.

Sine-Exponential Plot

Exp_fn Error Plot



Sine_fn Error Plot



SinExp_fn Error Plot

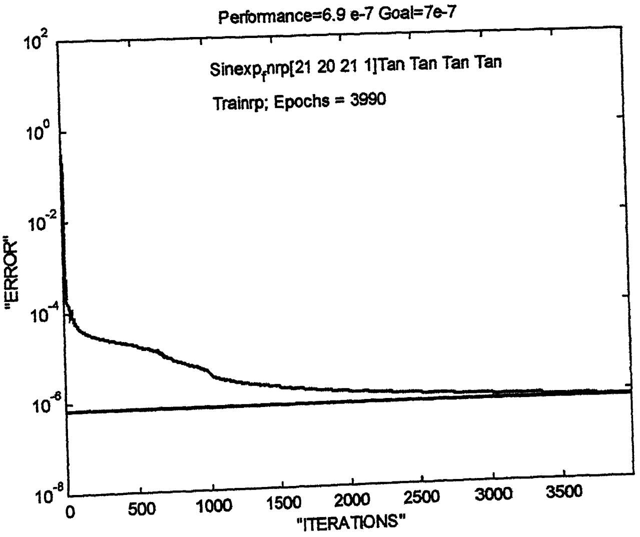
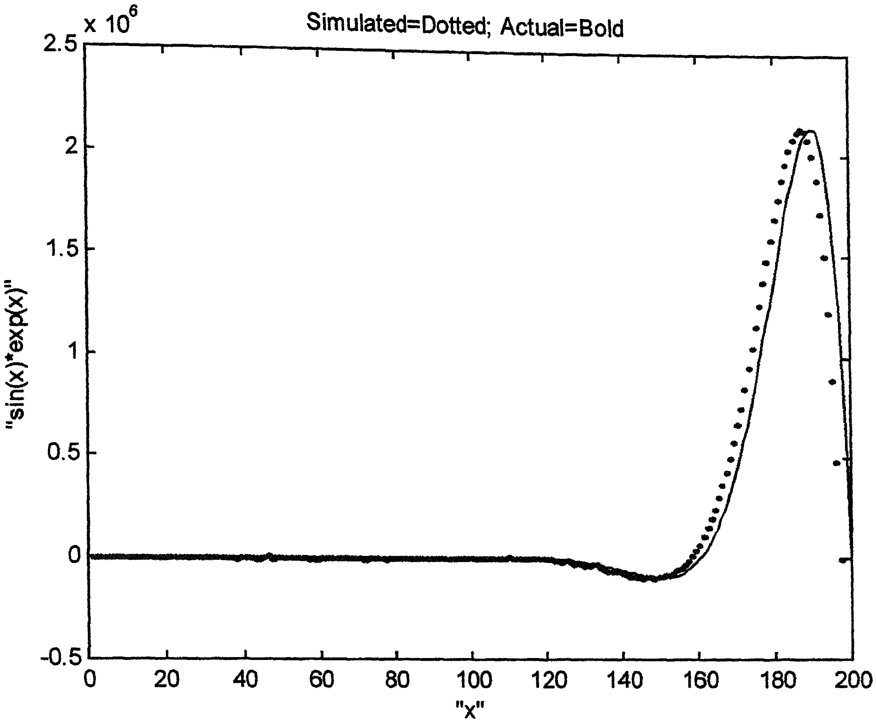


Figure 6.54a Function Approximation (y=sin(x)*exp(x))

Sine-Exponential Plot

Dotted=simulated; Bold=Actual



Simulated Vs Actual Output

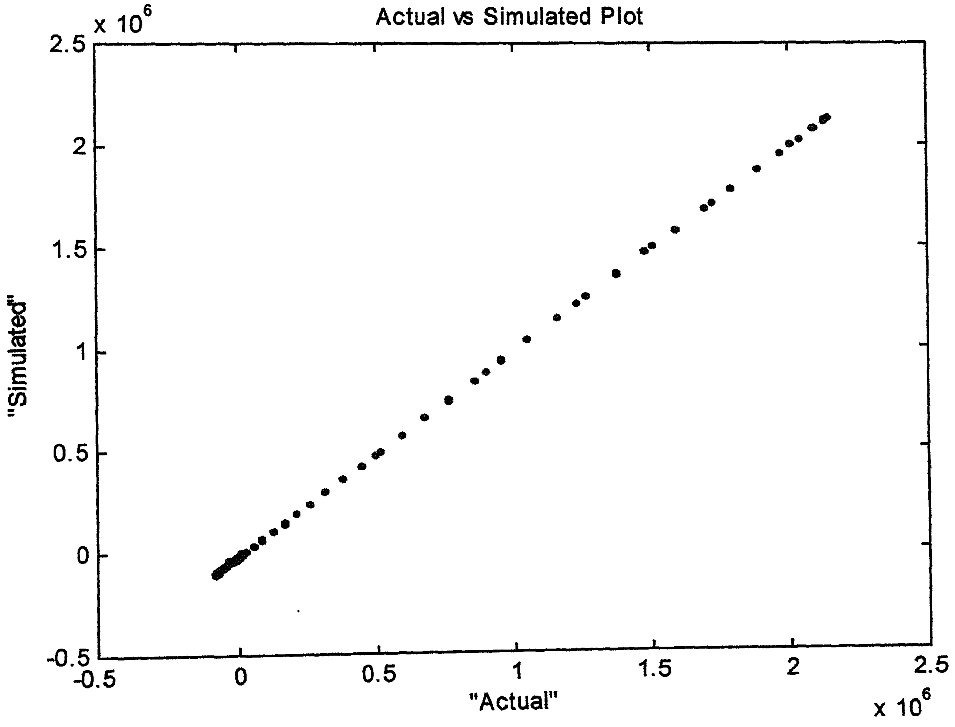
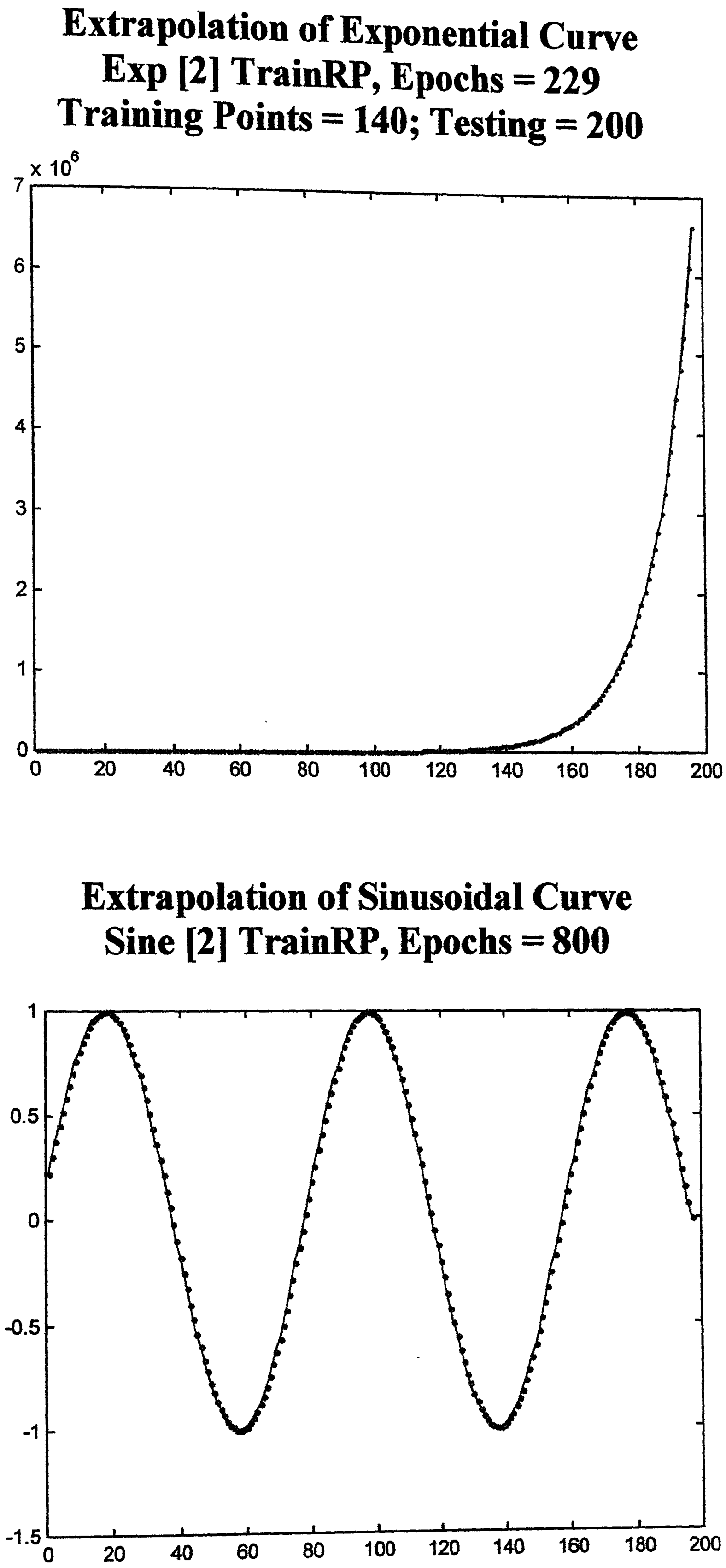


Figure 6.54b Function Approximation (y=sin(x)*exp(x))



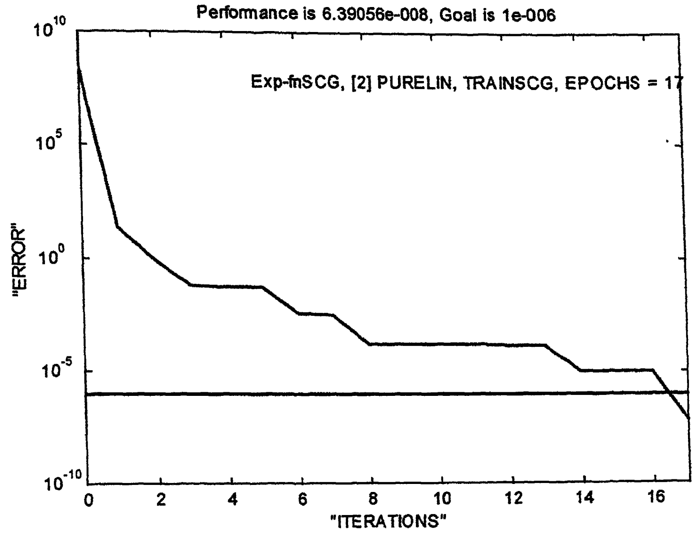
Extrapolation of Exponential Curve
Exp [2] TrainRP, Epochs = 229
Training Points = 140; Testing = 200

Extrapolation of Sinusoidal Curve
Sine [2] TrainRP, Epochs = 800

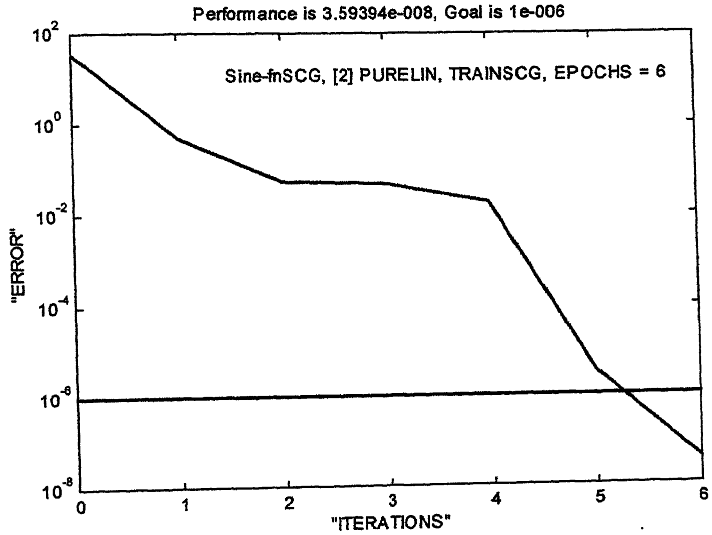
Figure 6.55: Extrapolation Curves

Sine-Exponential Plot

Exp_fn1 Error Plot



Sine_fn1 Error Plot



SinExp_fn1 Error Plot

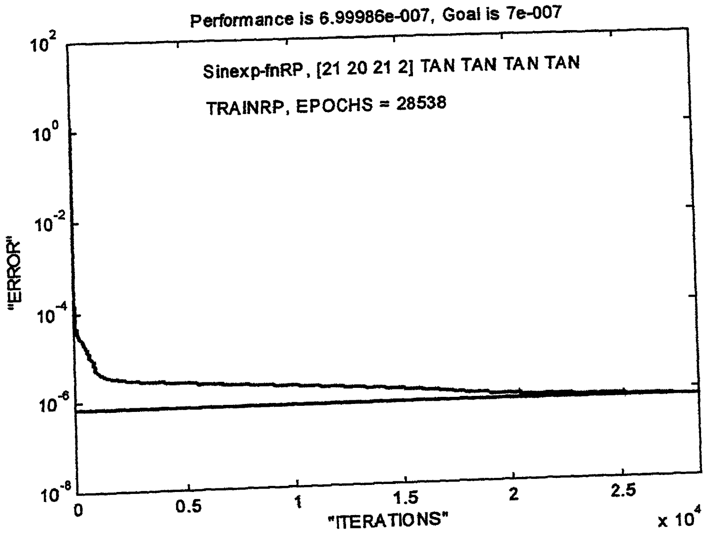
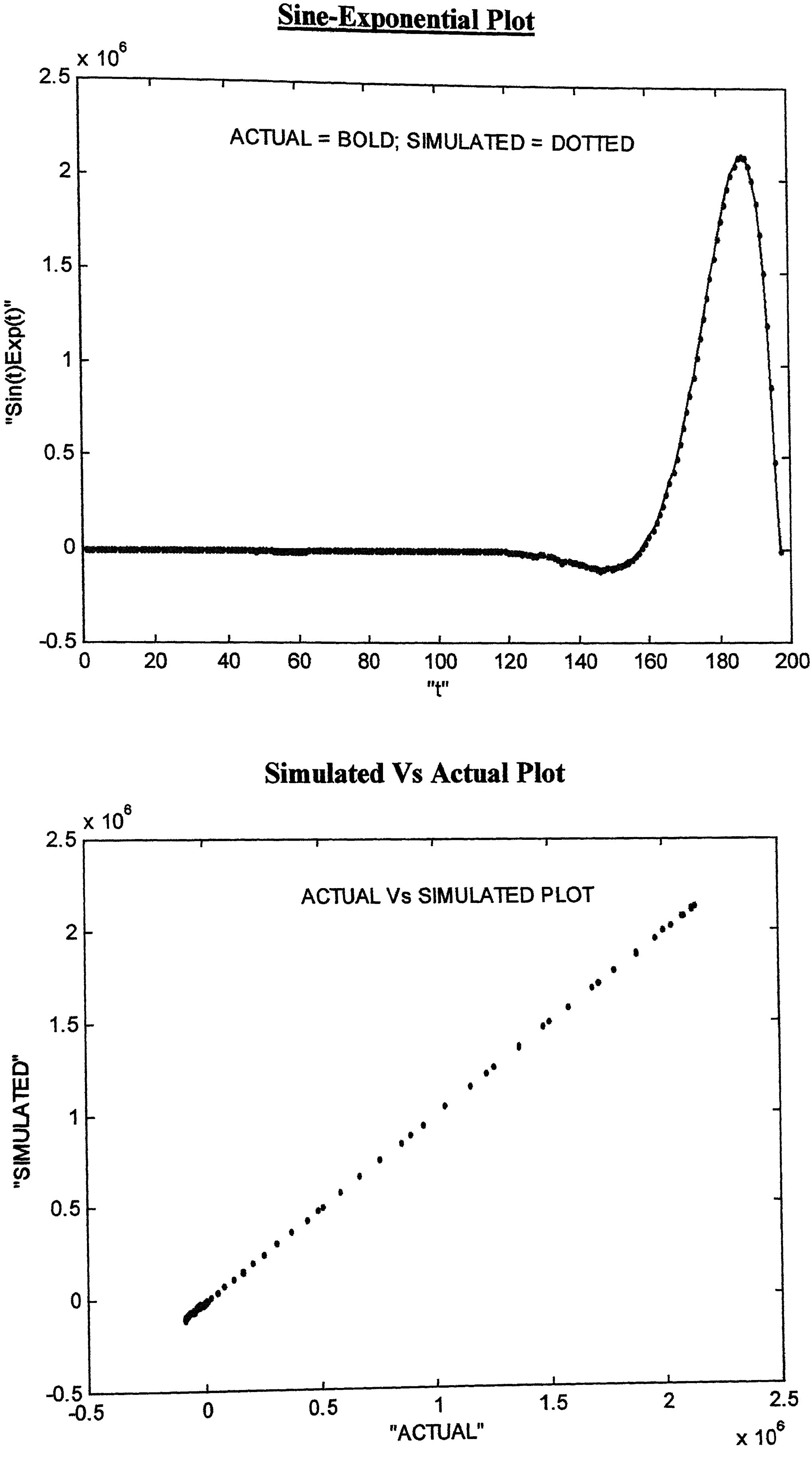


Figure 6.56a: Function Approximation (y = sin(x)exp(x))



Sine-Exponential Plot

Simulated Vs Actual Plot

Figure 6.56b: Function Approximation (y = sin(x)exp(x))

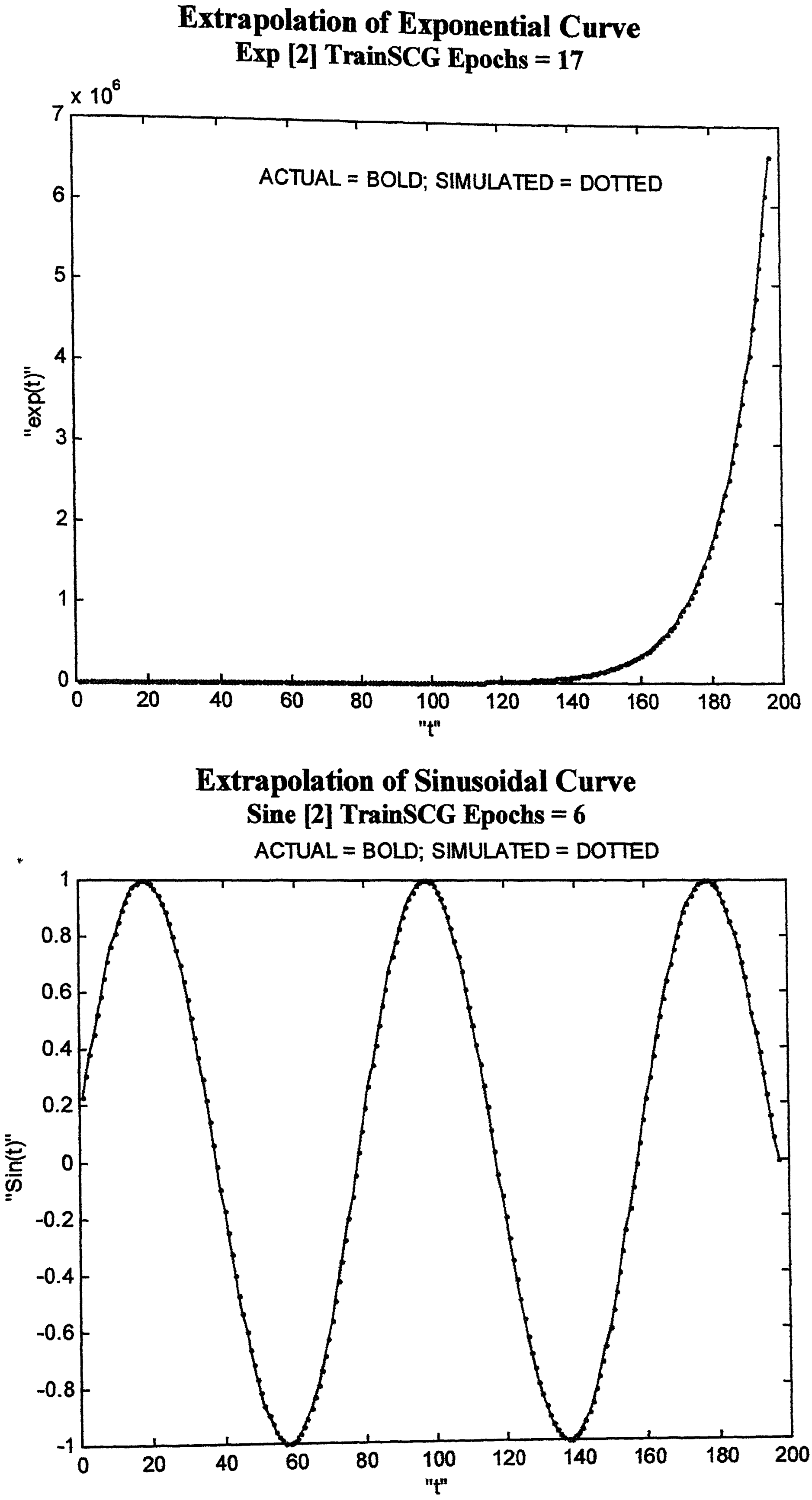
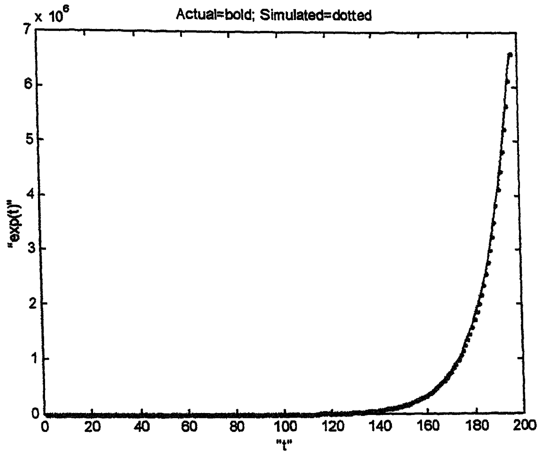


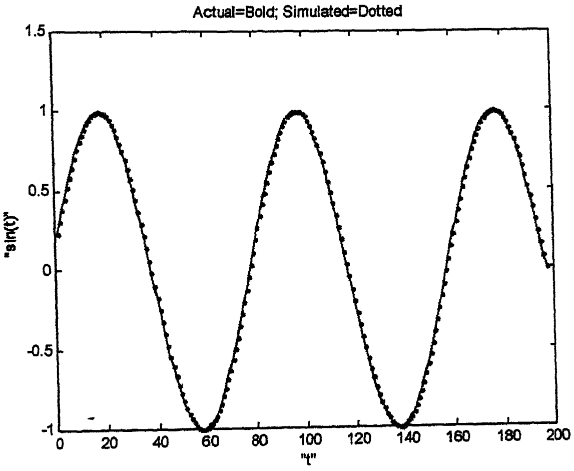
Figure 6.56c: Extrapolation Curves

JAVA PLOT

Extrapolation of Exponential Curve
Exp [1] {Purelin} TrainRP Epochs: 100000



Extrapolation of Sinusoidal Curve
Sin [1] {Purelin} TrainRP Epochs: 100000



Sinexp_fn1 [21 20 21 1] { Tansig Tansig Tansig Tansig}
TrainRP Epochs:97466

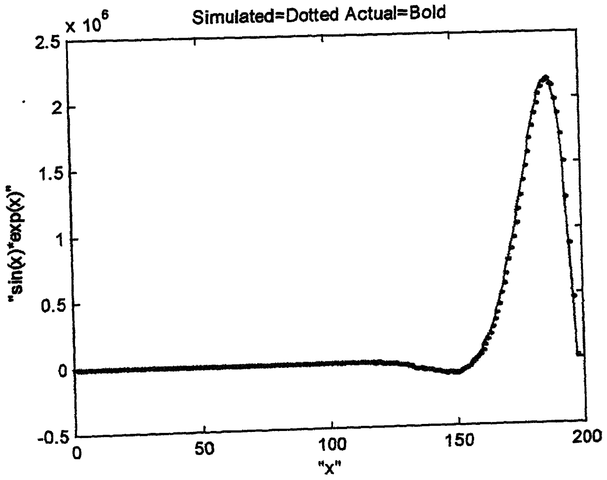


Figure 6.57: Function Approximation (y=sin(x)*exp(x))

References

1. Haykin S, "Neural Networks: A Comprehensive Foundation," New York, *Macmillan College Publishing Company,* 1994.

2. Zurada JM, "Introduction to Artificial Neural Systems," Delhi, *Jaico Publishing House,* 1994.

3. Schalkoff RJ, "Artificial Neural Networks," *McGraw-Hill International Editions,* 1997.

4. "Artificial Neural Networks for Intelligent Manufacturing", Edited by Cihan H Dagli.

5. Dan W. Patterson, "Artificial Neural Networks – Theory and Application", *Prentice Hall, Singapore,* 1995.

6. Mohamad H. Hassoum, "Fundamentals of Artificial Neural Networks", *Prentice Hall of India,* 1998.

7. B. Yegnarayana, "Artificial Neural Networks", *Prentice Hall of India,* 1999.

[redacted]